Tampereen yliopisto

Väinö Teisko

C#- JA ANGULAR-OHJELMISTOJEN TESTAUS

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Lokakuu 2020

TIIVISTELMÄ

Väinö Teisko: C#- ja Angular-ohjelmistojen testaus
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Lokakuu 2020

Ohjelmistoprojektin kasvaessa ei saa unohtaa ohjelmistotestauksen merkitystä osana kehitysprosessia. Projektin kasvaessa ja monimutkaistuessa myös odottamattomien ohjelmistovirheiden mahdollisuus kasvaa. Ohjelmistotestaamisen avulla pyritään löytämään mahdollisia ohjelmistovirheitä ja takaamaan ohjelmiston oikeellinen toiminta jo mahdollisimman varhaisesta kehitysasteesta lähtien. Ilman ohjelmistotestausta ohjelmistoon voi suuremmalla mahdollisuudella jäädä virheitä, jotka voivat pahimmillaan ilmetä vasta ohjelmiston julkaistussa versiossa loppukäyttäjän käytössä.

Tämän tutkielman tarkoituksena on kehittää John Deere Forestry Oy:n ohjelmistotestauskäytäntöjä ja antaa katsaus testaukseen liittyvään kirjallisuuteen ja hyviin käytäntöihin. Tutkielman laajuus on rajattu C#- ja Angular-ohjelmistojen testaamiseen, mutta ohjelmistotestaamista käsitellään aluksi yleisemmällä tasolla. Yleisemmän tason katsauksen jälkeen tutkielmassa vertaillaan erilaisia C#- ja Angular-ohjelmistojen testaamiseen käytettäviä testityökaluja. Lopuksi käydään läpi John Deere Forestry Oy:llä tehtyjä testityökaluihin liittyviä valintoja.

Avainsanat: C#, Angular, ohjelmistotestaus, yksikkötestaus, end-to-end-testaus, testikehys, jatkuva integraatio

Sisällys

1. Johdanto

Erilaisten ohjelmistoratkaisujen tarve on kasvanut viime vuosina. Nykyisin lähes kaikki yritykset haluavat mainostaa tai myydä tuotteitaan verkossa joko nettisivujen tai -sovellusten kautta. Tämä viime vuosina tapahtunut ohjelmistotuotannon kasvu on johtanut erilaisten ohjelmistokehitykseen liittyvien käytäntöjen syntymiseen ja kehittymiseen, joiden avulla pyritään nopeuttamaan, helpottamaan ja tehostamaan ohjelmistotuotannon prosesseja. Yksi näistä ohjelmistotuotannon prosesseista on ohjelmistotestaus.

Ohjelmistotestauksella tarkoitetaan ohjelmiston tai sen komponentin oikeellisen toiminnan tarkistamista. Ohjelmisto voidaan ajaa halutuilla syötteillä, mahdollisesti myös useamman kerran eri syötteillä, ja ajon tuloksia verrataan odotettuihin tuloksiin. Ohjelmistotestausta tehdään ohjelmiston laadun varmistamiseksi. Testauksen avulla pyritään säästämään aikaa ja resursseja, jotka ilman testausta voitaisiin joutua käyttämään *ohjelmistovirheiden* (bug) korjaamiseen ohjelmiston kehityksen myöhemmässä vaiheessa. Pahimmassa tapauksessa virheet voivat löytyä vasta, kun ohjelmisto on jo loppukäyttäjien käytössä. Testattavan ohjelmistokomponentin jokaista mahdollista syötettä ei useinkaan käytännössä voida testata, sillä siihen kuluisi liikaa aikaa. Kaikkien mahdollisten syötteiden testaamisen sijaan testaamisen tueksi on kehitetty erilaisia strategioita, joilla pyritään löytämään kaikki mahdolliset viat ohjelmistosta, mutta samalla minimoimaan tehtävien testiajojen määrää.

Ohjelmistoprojektin rakentumiseen lähdekoodista saattoi joskus menneisyydessä kulua jopa tunteja, joten ohjelmistokehittäjillä oli enemmän aikaa käytettävänä koodin manuaaliseen läpikäymiseen. Nykyisin koodiin tehtyjä muutoksia voi testata lähes välittömästi ja ohjelmistoista on tullut monella tavalla monimutkaisempia. Kehityssyklin nopeutuessa myös testausmenetelmiä on tarvinnut nopeuttaa. Ohjelmistotestausta voidaan tehdä monin eri tavoin riippuen esimerkiksi käytettävästä lähestymistavasta, testattavasta ohjelmistosta tai testiympäristöstä.

Tämän työn tavoitteena on kehittää John Deere Forestry Oy:n ohjelmistotestauskäytäntöjä ja antaa katsaus testaukseen liittyvään kirjallisuuteen ja hyviin käytäntöihin. Testattavat ohjelmistot rajattiin tämän työn puitteissa Microsoftin Visual Studio -kehitysympäristössä toteutettuihin C#-kielisiin ohjelmistoihin ja Angular-ohjelmistoihin.

Tutkielman luvussa 2 ohjelmistotestausta käsitellään ensin yleisemmin. Luvussa esitellään erilaisia lähestymistapoja testaamiseen, testauksen tasoja, testidatan luomismenetelmiä, testivetoista ohjelmistokehitystä ja jatkuvaa integraatiota.

Luvussa 3 keskitytään C#-ohjelmistojen testaamiseen. Luvussa vertaillaan joitain käytetyimmistä C#-yksikkötestikehyksistä Visual Studio -kehitysympäristössä. Lopuksi esitellään myös Visual Studion IntelliTest-työkalua.

Luvussa 4 keskitytään Angular-ohjelmistojen testaamiseen ja vertaillaan testaamiseen käytettäviä työkaluja. Käsiteltäviin työkaluihin sisältyy testikehyksiä, assertiokirjastoja, yksikkötestiajureita ja end-to-end-testiajureita.

Luvussa 5 Angular-testaamista tarkastellaan tarkemmin testityökalujen käyttöesimerkkien avulla. Esimerkit sisältävät monia luvussa 4 käsitellyistä Angular-testityökaluista.

Luvussa 6 käydään läpi John Deere Forestry Oy:llä tehtyjä valintoja C#- ja Angular-ohjelmistojen testaamiseen käytettäviin testityökaluihin ja testiautomaatioon liittyen. Työn lopusta löytyy vielä yhteenveto ja viiteluettelo.

2. Testauksesta yleisesti

Tässä luvussa käsitellään erilaisia lähestymistapoja testaamiseen, testausmenetelmiä, menetelmiä testidatan luomiseen, testivetoista ohjelmistokehitystä ja jatkuvaa integraatiota.

2.1. White box ja black box

White box –testaamisella tarkoitetaan testaamista, jossa testin kirjoittajalla on pääsy järjestelmän dokumentaatioon tai hän tuntee ohjelmiston lähdekoodin ja osaa siten luoda testejä, joilla saadaan suoritettua kaikki mahdolliset suorituspolut koodissa. Testitapauksia täytyy kuitenkin olla tarpeeksi paljon ja niiden tulee olla tarpeeksi erilaisia, jotta testit kattavat kaikki suorituspolut.

White box -testaamista käytetään tyypillisesti testauksen alkuvaiheessa, kun ohjelmoija itse vastaa testien suunnittelusta ja toteutuksesta [Watkins and Mills 2011]. White box -testejä ovat esimerkiksi yksikkötestit. Yksikkötestausta käsitellään tarkemmin kohdassa 2.2.1.

Black box –testaamisella taas tarkoitetaan testaamista, jossa testin kirjoittaja ei tunne lähdekoodia. Tällöin testin kirjoittaja joutuu testaamaan ohjelmistoa vain ohjelmiston ulkoiseen käytökseen pohjautuen. Black box -testaamista voidaan tarvita esimerkiksi vanhojen ohjelmistojen testaamiseen, joille ei ole saatavilla dokumentaatiota, tai kolmansien osapuolten kehittämien kaupallisten valmisohjelmistojen testaamiseen. Jos ohjelmiston vaatimusdokumentaatiota ei ole saatavilla, testien tulisi pohjautua joko ohjelmiston käyttöoppaisiin tai sen toimintaa kuvaaviin dokumentteihin [Watkins and Mills 2011].

Black box -testaamista käytetään tyypillisesti testauksen loppuvaiheessa, kuten hyväksyntätestauksessa. Monesti testien kirjoittajat käyttävät white box ja black box -tekniikoiden sekoitusta testien suunnittelussa, tällaista lähestymistapaa kutsutaan grey box -testaamiseksi. Tässä työssä käsitellään lähinnä white box -tasolla testaamista.

2.2. Testausmenetelmiä

Tässä osassa käydään läpi testauksen neljä tasoa: yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksyntätestaus. Lisäksi käsitellään regressiotestausta ja end-to-end-testausta, sillä ne ovat myös tärkeitä ohjelmistotestauksen muotoja.

2.2.1. Yksikkötestaus

Yksikkötestauksella tarkoitetaan yksittäisen ohjelmistokomponentin testausta erillään muista ohjelmistokomponenteista, tarkoituksena löytää virheitä komponentin logiikasta. Testit tulee kirjoittaa vastaamaan yksikölle asetettuja toimintavaatimuksia, jotta saadaan selville vastaako komponentin toiminta näitä vaatimuksia. Yksikkötestattavan komponentin laajuus voi vaihdella, mutta useimmiten yksikkötestattava komponentti on yksi luokka tai funktio. Luokkaa, joka määrittelee olion, voidaan testata esimerkiksi alustamalla kyseinen olio, muuttamalla olion tilaa ja tarkistamalla olion tilan muutos.

Ohjelmoijat voivat yksikkötestata omaa koodiansa ohjelmoinnin lomassa. Tällöin testaus tapahtuu white box –tasolla, sillä ohjelmoijat tuntevat oman koodinsa toiminnan. Yksikkötestaus suoritetaan yleensä tällä tasolla. Yksikkötestit voivat toimia hyvänä dokumentaationa myös muille kehittäjille, sillä testit ovat toimivia esimerkkejä komponentin käytöstä. Testien kirjoittajien täytyy kuitenkin osata kirjoittaa selkeitä testejä ja testejä täytyy päivittää komponenttien suunnittelun ja toiminnallisuuden mahdollisesti muuttuessa.

Järjestelmän yksikkötestattavuutta voi parantaa suunnittelemalla järjestelmän mahdollisimman *löyhästi kytketyksi* (loosely coupled), käyttäen uusimpia suunnittelumalleja, jotka maksimoivat löyhän kytkennän järjestelmässä [Garofalo 2011]. Löyhästi kytketyn järjestelmän komponentit eivät tunne järjestelmän muita komponentteja tai niiden toimintaa, vaan toimivat omina itsenäisinä kokonaisuuksinaan. Löyhästi kytkettyjä osia voi testata toisistaan erillään, sillä ne eivät vaadi järjestelmän muita osia toimiakseen. Löyhästi kytketyn komponentin sisäistä toteutusta voi myös helposti muuttaa, kunhan sen rajapinnat säilyvät eheinä.

2.2.2. Integraatiotestaus

Kun komponentti on yksikkötesteissä todettu toimivaksi, se on valmis testauksen seuraavaan vaiheeseen. Yksikkötestauksen jälkeen yksittäisiä ohjelmistokomponentteja integroidaan toimimaan yhdessä osajärjestelminä. Vaikka ohjelmistokomponenteissa ei olisikaan löytynyt ongelmia yksikkötesteissä, komponenttien yhteistoiminta ei silti välttämättä toimi odotetusti. *Integraatiotestauksessa* testataan yksittäisten ohjelmistokomponenttien rajapintoja ja yhteensopivuutta toistensa kanssa osajärjestelmänä, kun komponentit on ensin yksikkötestattu. Integraatiotestit suoritetaan yleensä white box -tasolla, mutta ne voidaan suorittaa myös black box -tasolla.

Esimerkkinä yksikkötestien onnistumisesta, mutta komponenttien yhteistoiminnan epäonnistumisesta Black [2011] nostaa esille Yhdysvaltojen 1990-luvulla suorittaman Mars-tehtävän, joka epäonnistui osajärjestelmien yhteensopivuusongelman vuoksi. Yksi navigaatioon liittyvä osajärjestelmä käytti metrijärjestelmää, kun taas toinen käytti englantilaisia mittausyksiköitä. Käytettyjen mittayksiköiden eroavaisuus johti luotaimen syöksyyn Marsin pinnalle liian suurella nopeudella. Yksikkötesteissä tätä osajärjestelmien yhteensopivuusongelmaa ei havaittu, sillä osajärjestelmät toimivat erillään. Integraatiotestit ovat tärkeä osa ohjelmistotestausta, tämäkin virhe olisi voitu huomata integraatiotesteissä.

Integraatiotesteissä havaitut ongelmat voidaan myös löytää järjestelmätesteissä, mutta ongelmat muutaman osajärjestelmän välillä on helpompi eristää testattaessa pienempää osaa järjestelmästä, kuin kokonaista sadoista tai jopa tuhansista osajärjestelmistä koostuvaa järjestelmää. Järjestelmätestausta käsitellään tarkemmin kohdassa 2.2.3. Black [2011] esittelee neljä erilaista lähestymistapaa integraatiotestaukseen:

1. *Big bang -testaus*: kaikki integroitavat komponentit testataan yhdessä ilman testiajureita, mock-olioita tai tynkiä. Tämä lähestymistapa on halpa ja nopea toteuttaa, sillä se vaatii vain yhden testin, jossa kaikki integroitavat komponentit on heti integroitu yhdeksi. Ongelmia kuitenkin löytyy:
    - Mahdollisten virheiden alkuperän paikantaminen on sitä vaikeampaa, mitä enemmän komponentteja testissä on mukana. Big bang -testissä tämä ongelma on pahimmillaan, sillä siinä on mukana kaikkien komponenttien toteutukset. Parhaiten tämä lähestymistapa toimii pienemmissä järjestelmissä, joissa ei ole montaa yhdistettävää komponenttia.
    - Kaikkien komponenttien testaus yhdessä ei myöskään välttämättä ole nopeampaa kuin useamman pienemmän integraatiotestin kirjoittaminen käyttäen testiajureita, mock-olioita ja tynkiä.
    - Kaikkien komponenttien täytyy olla valmiita integraatiotestiin. Pienempiä integraatiotestejä voisi kirjoittaa jo ennen kuin kaikki komponentit ovat integraatiotestauskelpoisia. Virheet koodissa tulisi löytää mahdollisimman aikaisessa kehitysvaiheessa, joten integraatiotestauksen aloittamista ei välttämättä kannata viivyttää.
2. *Alhaalta ylös -testaus* (Bottom-up testing): ohjelmistoa integroidaan yhteen taso kerrallaan, lähtien pohjimmaiselta tasolta, kunnes päästään täysin integroituun järjestelmätestiin. Alimmainen taso voi olla esimerkiksi tietokannan tai laitteiston käyttökerros ja ylin taso on kokonainen järjestelmä. Tasot voidaan integroida kokonaisina tai osissa riippuen tason laajuudesta ja virhealttiudesta.
    - Tässä lähestymistavassa käytetään *testiajureita*, joiden avulla alemman tason komponentteja kutsutaan testin yhteydessä. Ajurit imitoivat ylemmän tason komponentteja, jotka valmiissa järjestelmässä kutsuvat alemman tason komponentteja, mutta ajurien avulla voidaan testata alempaa tasoa ennen kuin ylempää tasoa on kehitetty valmiiksi. Kun ylempi taso integroidaan mukaan testiin, aiemmat ajurit korvataan tällä ylemmällä tasolla ja taas seuraavan ylemmän tason testaamiseksi luodaan uudet ajurit. Tasoja integroidaan yhteen, kunnes päädytään täysin integroituun järjestelmätestiin.
    - Löydetyt virheet on helppo eristää, sillä komponentit integroidaan yhteen osissa kerroksittain.
    - Jos ylimmillä tasoilla on virheitä, niitä ei löydetä ennen viimeisiä integrointikierroksia. Jos ylemmän tason virheet vaativat muutoksia alempiin tasoihin, kaikki välissä olevat tasot pitää testata uudelleen.
3. *Ylhäältä alas -testaus* (Top-down testing): samankaltainen kuin alhaalta ylös -testaus, mutta lähtien ylimmästä tasosta ja edeten pohjimmaiselle tasolle. Ohjelmistoa integroidaan yhteen taso kerrallaan, kunnes koko järjestelmä on integroitu

yhteen. Tässäkin lähestymistavassa tasot voidaan integroida kokonaisina tai osissa, riippuen niiden monimutkaisuudesta ja virhealttiudesta.

- Alhaalta ylös -lähestymistavassa käytettävien ajurien korvikkeena ylhäältä alas -testauksessa käytetään *tynkiä*. Tyngillä simuloidaan alempien tasojen toiminnallisuutta, kuten ajureilla simuloidaan ylempiä tasoja. Tynkien avulla voidaan testata ylempää tasoa, vaikka alempaa tasoa ei olisi vielä kehitetty valmiiksi. Tyngät sijoitetaan alempien tasojen funktioiden kutsujen tilalle, jotta alempien tasojen toteutusta ei käytettäisi testissä. Kun alempi taso integroidaan mukaan testiin, tyngät korvataan alemman tason funktiokutsuilla ja alemmalle tasolle luodaan uudet tyngät, ennen kuin myös seuraava taso integroidaan mukaan. Kuten alhaalta ylös -lähestymistavassakin, tasoja integroidaan yhteen, kunnes päädytään täysin integroituun järjestelmätestiin.
- Virheet on helppo eristää, sillä integrointi tehdään kerroksittain. Kuitenkin jos alimmilla tasoilla löytyy virheitä, ne löydetään vasta viimeisillä integrointikierroksilla. Alimman tason virheet voivat johtaa myös ylempien tasojen muutoksiin, jolloin myös välissä olevat tasot tulee testata uudelleen.

4. *Selkärankatestaus* (Backbone testing): integroinnin järjestystä ei päätetä järjestelmän rakenteen mukaan, vaan liiketoiminnallisen ja teknisen riskin perusteella. Kriittisimmät komponentit integroidaan ensimmäiseksi selkärangaksi. Selkäranka voi vaatia ajureita ja tynkiä molempia, sillä selkäranka ei välttämättä ole järjestelmän pohjimmaisella tai päällimmäisellä tasolla. Selkärankaan integroidaan vain siihen liittyviä komponentteja joko sen ajurien tai tynkien tilalle, tärkeysjärjestyksessä.

2.2.3. Järjestelmätestaus

Integraatiotestauksen jälkeen siirrytään testaamaan kokonaista järjestelmää. *Järjestelmätestauksella* tarkoitetaan kokonaisen, täysin integroidun järjestelmän testaamista yhtenä kokonaisuutena. Toisin kuin yksikkö- ja integraatiotestit, järjestelmätestit voidaan toteuttaa myös black box –tasolla. Silloin testaajalla ei ole pääsyä ohjelmiston sisäiseen toimintaan, vaan vuorovaikutus tapahtuu pelkästään ohjelmiston käyttöliittymän kautta. Järjestelmätestauksessa ei enää olla kiinnostuneita yksittäisten komponenttien tai osajärjestelmien toiminnasta tai rajapinnoista, vaan järjestelmää tarkastellaan pelkästään yhtenä kokonaisuutena.

Järjestelmätestauksen tuloksia voidaan verrata ohjelmiston funktionaalisiin ja liiketoiminnallisiin vaatimuksiin, sillä järjestelmätestissä testataan jo valmista ohjelmistojärjestelmää. Ohjelmistosta tarkistetaan, että se ei kaadu ja että sen toiminta vastaa sille asetettuja toiminnallisia vaatimuksia.

2.2.4. Hyväksyntätestaus

Hyväksyntätestaus on ohjelmiston viimeinen testausvaihe yksikkö-, integraatio- ja järjestelmätestauksien jälkeen. Ohjelmistoa testataan käyttäjien tarpeiden ja vaatimusten kannalta. Hyväksyntävaatimukset voivat olla kirjattuna erilliseen dokumenttiin jo projektin alussa. Hyväksyntävaatimukset ovat asiakkaan määrittelemiä ja ne ohjaavat ohjelmiston kehitystä.

Hyväksyntätestaus suoritetaan yleensä joko asiakkaalla tai loppukäyttäjillä, jotka päättävät onko ohjelmisto käyttöönottovalmis. Toisin kuin järjestelmätestauksessa, hyväksyntätestauksen pääpaino on loppukäyttäjän mielipiteellä ohjelmiston toiminnasta, eikä vain funktionaalisilla vaatimuksilla. Käyttäjätestausta kutsutaan myös beetatestaukseksi. Hyväksyntätestauksessa käytetään black box –testausmetodia, eli testaaja ei näe ohjelmiston sisäistä toteutusta.

Jos yksikkötesteillä varmistetaan, että koodi tekee juuri mitä kehittäjä haluaa sen tekevän, niin hyväksyntätesteillä varmistetaan, että koodi tekee juuri mitä käyttäjä haluaa sen tekevän. [Sale 2014]

2.2.5. Regressiotestaus

Regressiolla tarkoitetaan ohjelmistoon tehdyn muutoksen seurauksena syntynyttä ohjelmistovirhettä, jonka takia jokin aiemmin toiminut ominaisuus lakkaa toimimasta oikeellisesti. *Regressiotestauksella* pyritään varmistamaan, että muutokset ohjelmiston koodissa eivät ole tarkoituksettomasti vaikuttaneet ohjelmiston olemassa olevaan toiminnallisuuteen. Regressiotestit koostuvat aiemmin ajetuista testeistä, joilla vanhan toiminnallisuuden oikeellista toimintaa on alun perin testattu. Vanhat testit ajetaan uudelleen regressiotesteinä, jotta saadaan varmistettua, toimiiko olemassa oleva toiminnallisuus samalla tavalla kuin aiemminkin, vai ovatko uudet koodimuutokset hajottaneet vanhaa toiminnallisuutta. Regressiot voidaan jakaa kolmeen päätyyppiin:

1. *Paikallinen regressio* (local regression): muutoksen seurauksena syntyy uusi virhe.
2. *Etäregressio* (remote regression): muutos järjestelmän yhdessä osassa hajottaa toiminnallisuuden toisessa osassa.
3. *Paljastettu regressio* (unmasked/exposed regression): muutoksen seurauksena paljastuu jo olemassa ollut virhe. [Black 2011]

Regressioiden löytämiseksi käytetään myös erilaisia regressiotestausstrategioita. Jos testit ovat hyvin suunniteltuja kaikkien mahdollisten regressioiden testaamiseen, toistamalla kaikki testit voidaan olla melko varmoja siitä, että uusia regressioita ei ole syntynyt [Black 2011]. Regressiotestaamiseen on kuitenkin kehitetty tehokkaampiakin menetelmiä, kuin kaikkien testien jatkuva uudelleensuoritus.

Toistamalla vain osa testeistä voidaan säästää testien suorittamiseen kuluvia resursseja, jos olennaisten testien valinta ja valittujen testien suorittaminen on edullisempaa

kuin kaikkien testien suorittaminen. Black [2011] esittelee kolme tapaa valita uudelleensuoritettavat testit regressiotestauksessa:

1. Jäljitettävyys: Jäljitä mihinkä vaatimuksiin, suunnitteluelementteihin tai laaturiskeihin ohjelmistoon tehty muutos vaikuttaa, ja valitse niihin liittyvät testit uudelleensuoritettavaksi. Testit tulisi pyrkiä liittämään tiettyihin vaatimuksiin, suunnitteluelementteihin tai koodin osiin niiden jäljitettävyyden takaamiseksi [Craig and Jaskiel 2002].
2. Muutosanalyysi: Tutkimalla järjestelmän rakennetta selvitä kuinka kauas järjestelmän muihin osiin koodiin tehdyt muutokset voivat vaikuttaa ja valitse näihin osiin liittyvät testit suoritettavaksi. Tämä lähestymistapa vaatii syvällistä tuntemusta järjestelmän rakenteesta ja toiminnasta.
3. Laaturiskianalyysi: Korkean liiketoiminnallisen riskin omaavat ohjelmiston osat tulisi uudelleen testata aina. Tärkeyden vuoksi tällaiset ohjelmiston osat tulisi uudelleen testata, vaikka ohjelmistomuutokset eivät jäljitettävyyden tai muutosanalyysin mukaan vaikuttaisikaan näihin ohjelmiston osiin. Aina on olemassa pieni mahdollisuus, että jokin virhe on jäänyt huomaamatta.

Priorisoimalla testien suoritusjärjestystä voidaan tehostaa testien suorituskykyä jollain tietyllä mittapuulla. Testit voidaan esimerkiksi suorittaa sellaisessa järjestyksessä, jossa ohjelmiston jonkin tietyn osan virheet testataan ensin [Rothermel *et al.* 2001]. Näin saadaan tietoa kyseisen osan virheistä nopeammin, kuin jos se testattaisiin vasta viimeisenä. Myös regression riskinlieventämiseen voidaan käyttää erilaisia menetelmiä [Black 2011]:

1. Päivitysten julkaiseminen hitaammalla syklillä: Suuremmat julkaisut yleensä testataan perusteellisemmin ja pidempi aika julkaisujen välillä antaa myös enemmän aikaa ajaa useampia testejä.
2. Asiakas- tai käyttäjätestaus: käyttäjätestauksella voidaan löytää suuri osa ohjelmiston virheistä. Julkinen käyttäjätestaus ei kuitenkaan korvaa muodollista testausta, sillä käyttäjät eivät useinkaan suorita kovin laajoja testejä, jos heille annetaan vapaat kädet testattavaan ohjelmistoon ilman testitapauksia [Craig and Jaskiel 2002].
3. Rajoita kiireelliset hätäpäivitykset vain niitä tarvitseville: kiireellisiä hätäpäivityksiä ei aina testata tarpeeksi perusteellisesti niiden kiireellisyyden takia. Hätäpäivitykset voidaan joissain tapauksissa jakaa vain niitä tarvitseville käyttäjille, jolloin regressioriski rajoittuu vain osaan käyttäjistä. Hätäpäivitykset voidaan myöhemmin liittää osaksi suurempia, paremmin testattuja päivityksiä, jolloin regressioriskin kasvu osalle käyttäjistä on vain väliaikainen.

2.2.6. End-to-end-testaus

End-to-end-testaaminen on testaamisen muoto, jossa testataan sovelluksen työnkulkua loppukäyttäjän näkökulmasta. Suoritettavat testitapaukset luodaan sellaisiksi, että ne simuloivat todellisia käyttötilanteita. Testit kirjoitetaan vastaamaan asiakkaan järjestelmälle asettamia toiminnollisia vaatimuksia.

End-to-end-testeissä automatisoidaan vuorovaikutusta sovelluksen käyttöliittymän kanssa, jotta testi vastaisi oikean käyttäjän toimintoja sivulla. Testissä voidaan esimerkiksi avata verkkosivu, kirjoittaa syötteitä sivulla olevaan syötekenttään ja klikata painiketta, jonka jälkeen testin lopuksi tarkistetaan sivun oikeellinen reagointi sivulla suoritettuihin toimenpiteisiin. End-to-end-testeissä ei siis kutsuta testattavan ohjelmiston funktioita suoraan, vaan sen sijasta käsitellään käyttöliittymäelementtejä, kuten oikea loppukäyttäjäkin tekisi.

2.3. Testidatan luonti

Testidatalla tarkoitetaan testattavalle ohjelmistolle testin yhteydessä annettavia syötteitä. Syötteitä käytetään joko *positiiviseen testaamiseen* tai *negatiiviseen testaamiseen*. Positiivisessa testaamisessa testataan tuottaako syöte oikeellisen tuloksen. Negatiivisessa testaamisessa testataan ohjelmiston reaktioita poikkeuksellisiin tai odottamattomiin syötteisiin. Testidata täytyy suunnitella hyvin, jotta testit kattavat kaikki mahdolliset tilanteet. Hyvin suunnitellut testit vähentävät ohjelmistossa piilevien virheiden mahdollisuutta, joka taas parantaa ohjelmiston laatua.

White box -testaamisessa testidata tulisi mielellään luoda sellaiseksi, että kaikki koodin suorituspolut testattaisiin ainakin kerran. Näin saadaan selville, että kaikki eri suorituspolut toimivat ainakin jollain tasolla. Black box -testaamisessa ohjelmiston kaikkien suorituspolkujen testaamista ei voida varmistaa, sillä lähdekoodi ei ole näkyvillä testaajalle. Testidatan luomista voi myös vaikeuttaa mahdollinen ohjelmiston toimintaa kuvaavan dokumentaation vajaus tai puute, varsinkin black box -testaamisessa.

Ekvivalenssiluokittelussa testattavan komponentin syötteet ja tulosteet luokitellaan luokiksi. Luokat luodaan siten, että komponentti käsittelee kaikki samaan luokkaan kuuluvat arvot samalla tavalla. Ekvivalenssiluokittelun periaate on, että luokasta voidaan testata yhtä arvoa ja se vastaa kaikkien luokan arvojen testaamista. Haasteet piilevät sellaisten syöte- ja tulosteryhmien tunnistamisessa, joilla kaikki erilaiset tilanteet saadaan testattua. [Watkins and Mills 2011]

Raja-arvoanalyysi on ekvivalenssiluokittelun kaltainen tekniikka testidatan luomiseen. Ekvivalenssiluokittelussa luokkia edustavat testiarvot valitaan luokan sisältä, kun taas raja-arvoanalyysissä arvot valitaan luokkien ulkoreunoilta [Watkins and Mills 2011]. Raja-arvoanalyysiä ja ekvivalenssiluokittelua voidaan käyttää yhdessä luodessa testidataa. Ekvivalenssiluokittelulla ensin tunnistetaan arvoluokkien rajat ja sen jälkeen raja-arvoanalyysillä valitaan testiarvot arvoluokkien reunoilta.

Oletetaan, että oltaisiin testaamassa järjestelmää, joka luokittelee käyttäjän ikäryhmään käyttäjän syöttämän iän perusteella. Käyttäjät luokitellaan seuraavien sääntöjen perusteella:

- Jos käyttäjän ikä on negatiivinen, näytä virheviesti.
- Jos ikä on positiivinen ja enintään 12 vuotta, käyttäjä on ”lapsi”.
- Jos käyttäjä ei ole ”lapsi” ja ikä on enintään 18, käyttäjä on ”nuori”.
- Jos käyttäjä ei ole ”lapsi” eikä ”nuori” ja ikä on enintään 59, käyttäjä on ”aikuinen”.
- Jos käyttäjä ei ole ”lapsi” eikä ”nuori” eikä ”aikuinen”, käyttäjä on ”vanhempi aikuinen”.

Testiajo	Ikä	Testattu arvoluokka	Odotettu tulos
1	-10	Ikä < 0	Virheviesti
2	10	0 <= Ikä <= 12	Lapsi
3	15	13 <= Ikä <= 18	Nuori
4	30	19 <= Ikä <= 59	Aikuinen
5	70	Ikä >= 60	Vanhempi aikuinen

Taulukko 1. Ekvivalenssiluokittelun avulla luodut testiajot.

Käyttäen ekvivalenssiluokittelua saadaan taulukon 1 mukaiset arvoluokat ja testiarvot. Ekvivalenssiluokittelu tuotti viisi eri arvoluokkaa, jotka vastaavat viittä eri tulosta. Jokaisesta arvoluokasta testataan vain yksi arvo, sillä kaikki yhden arvoluokan arvot antavat saman tuloksen.

Testiajo	Ikä	Testattu raja-arvo	Odotettu tulos
1	-1	0-1	Virheviesti
2	0	0	Lapsi
3	1	0+1	Lapsi
4	11	12-1	Lapsi
5	12	12	Lapsi
6	13	12+1	Nuori
7	17	18-1	Nuori
8	18	18	Nuori
9	19	18+1	Aikuinen
10	58	59-1	Aikuinen
11	59	59	Aikuinen
12	60	59+1	Vanhempi aikuinen

Taulukko 2. Raja-arvoanalyysin avulla luodut testiajot.

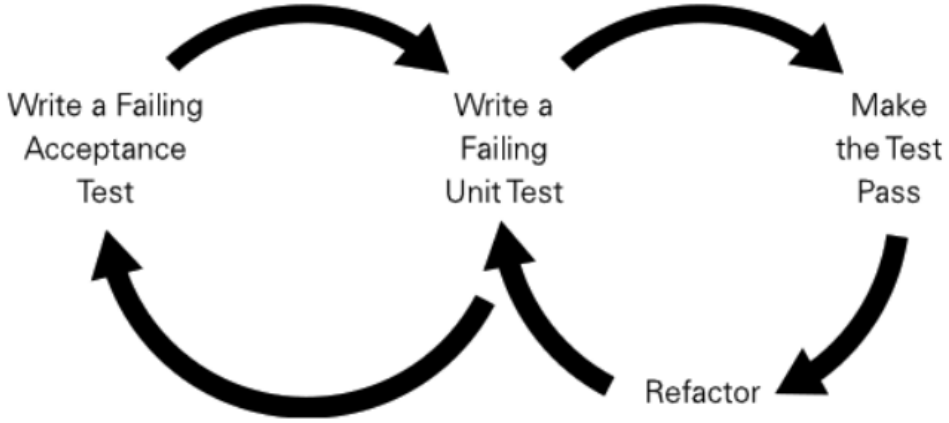
Raja-arvoanalyysillä saadaan taulukon 2 mukaiset raja-arvot ja testiarvot. Tunnistettujen raja-arvojen (0, 12, 18, 59) lisäksi testataan myös raja-arvon molemmin puolin olevia läheisiä arvoja, tässä tapauksessa oli luontevaa testata raja-arvot ±1.

2.4. Testivetoinen ja käyttäytymisvetoinen kehitys

Testivetoinen kehitys (Test Driven Development, TDD) on ohjelmistokehityksen prosessi, jossa ohjelmistoprojektin asiakkaan määrittämien vaatimusten pohjalta kirjoitetaan yksikkötestitapaukset ennen kuin kirjoitetaan yhtään koodia. Koodia kirjoitetaan vain vastaamaan ennalta määrättyjä testitapauksia, ei yhtään enempää. Testitapaukset ovat siis aina askeleen edellä koodia. Tämä helpottaa koodin kirjoitusta, sillä koodille on kirjoitettu yksikkötestit jo valmiiksi ja halutut toiminnot on siten dokumentoitu kooditasolla. Testivetoinen kehitys antaa varmuutta ohjelmiston toiminnasta, koska oikeellinen toiminta on helppo tarkistaa yksikkötestien kautta. Ongelmana on, että joidenkin testien kirjoittaminen ennen koodia voi olla haastavaa, jos haluttua toiminnallisuutta ei ymmärretä vielä tarpeeksi syvällisesti. Mahdollisia bugeja toistavia testejä on myös lähes mahdotonta kirjoittaa, ennen kuin bugien syyt ovat tiedossa. Testivetoisen kehityksen tehtävänä on lähinnä auttaa kehittäjää kirjoittamaan koodiin juuri oikeanlainen, testien määrittämä toiminnallisuus.

Hyväksymistestausvetoinen kehitys (Acceptance Test Driven Development, ATDD) taas on lähinnä kommunikaatiotyökalu kehittäjän, testaajan ja asiakkaan välillä. Ohjelmiston vaatimukset määritetään asiakkaan tasolla, jolloin kaikki sidosryhmät ymmärtävät niiden sisällön. Tästä kaikkien osapuolten ymmärtämästä esitysmuodosta vaatimukset voidaan hajottaa eteenpäin hyväksyntätesteiksi ja lopulta teknisempään muotoon pienemmiksi yksikkötesteiksi testivetoisen kehityksen mukaisesti. Hyväksymistestausvetoinen kehitys synnyttää asiakkaan toiminnollisia ja ei-toiminnollisia vaatimuksia yleisellä tasolla, kun taas testivetoinen kehitys auttaa kehittäjää kirjoittamaan koodin testeihin sopivalla tavalla.

Käyttäytymisvetoisessa kehityksessä (Behavior Driven Development, BDD) kirjoitetaan hyväksyntätestit ennen kuin kirjoitetaan yhtään koodia. Hyväksyntätestit voidaan kirjoittaa hyväksymistestausvetoisen kehityksen mukaisesti luotujen ylemmän tason vaatimusten pohjalta. Käyttäytymisvetoinen kehitys on lähellä hyväksymistestausvetoista kehitystä, mutta kun hyväksymistestausvetoisessa kehityksessä keskitytään vaatimusten määrittämiseen kaikkien sidosryhmien ymmärtämällä yhteisellä kielellä, käyttäytymisvetoisessa kehityksessä keskitytään ohjelmiston käyttäytymiseen tarkemmalla tasolla. Kun testivetoisessa kehityksessä painotetaan yksikkötestien luomista ennen koodia, käyttäytymisvetoisessa kehityksessä vastaavasti painotetaan hyväksyntätestien luomista ennen koodia [Sale 2014].



Kuva 1. Kehityssykli, kun yksikkötestaukseen lisätään myös hyväksyntätestaus. [Sale 2014]

Kun testivetoiseen kehitykseen otetaan mukaan myös käyttäytymisvetoinen kehitys, saadaan kuvan 1 mukainen kehityssykli. Ensin kirjoitetaan hyväksyntätestit, sitten yksikkötestit, yksikkötestit toteuttava koodi ja syklin lopuksi tehdään tarpeen mukaan koodin refaktorointia.

2.5. AAA-malli

Bill Wake tunnisti ja nimesi yksikkötestimetodien kirjoituksessa käytetyn *AAA-mallin* (”Arrange-Act-Assert”) vuonna 2001 [Beck 2002; Wake 2011]:

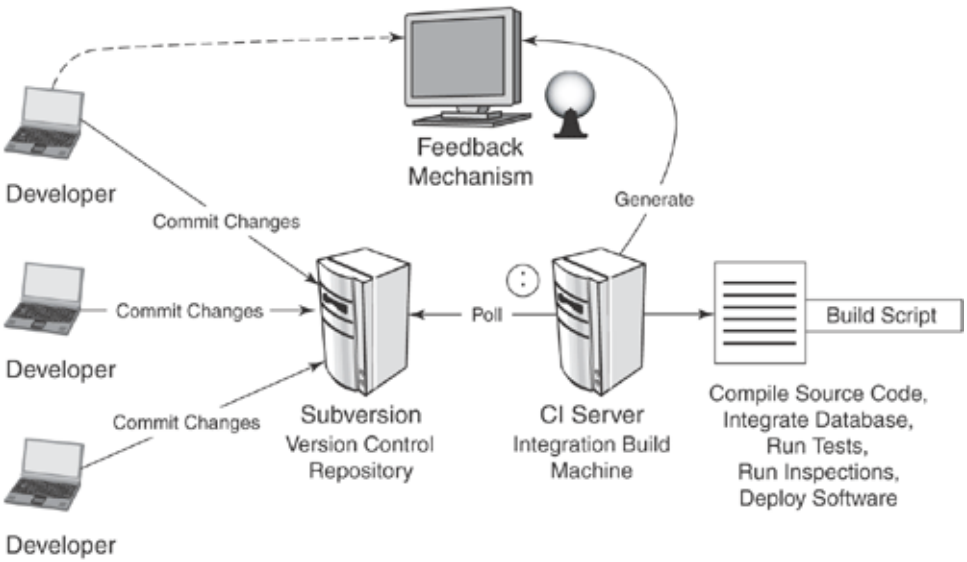
1. *Arrange*: alustetaan testattava olio ja/tai testiin tarvittavat muuttujat.
2. *Act*: suoritetaan testattava toimenpide.
3. *Assert*: tarkistetaan toimenpiteen tuloksen oikeellisuus.

AAA-mallin mukaan testin toiminnan oikeellisuus tulisi tarkistaa vasta testimetodin lopussa, alustuksen ja toimenpiteen jälkeen. Vaikka AAA-mallin mukaan yksikkötestimetodin tulisi alkaa testin tarvitsemien muuttujien alustuksilla (”Arrange”), Wake suosittelee aloittamaan yksikkötestin kirjoittamisen Assert-vaiheesta eli testin oikeellisuuden tarkistuksesta. Jos testi toimisi, kuinka saisit sen selville? [Wake 2011]

Assert-osasta lähtemistä yksikkötestejä kirjoittaessa voisi verrata testivetoiseen ohjelmistokehitykseen. Testivetoisessa kehityksessä kirjoitetaan ensin testit, sitten mahdollisimman vähän koodia testin läpäisemiseksi. Assert-lähtöisessä yksikkötestin kirjoituksessa kirjoitetaan ensin testin tuloksen oikeellisuuden tarkistus, sitten mahdollisimman vähän koodia testin toteuttamiseksi [Hill and Kerievsky] pysyen kuitenkin AAA-mallissa testin luettavuuden parantamiseksi.

2.6. Jatkuva integraatio

Jatkuvalla integraatiolla (Continuous Integration, CI) tarkoitetaan ohjelmistotuotannon käytäntöä, jossa kehittäjät yhdistävät jaetun projektin lähdekoodimuutoksensa yhdeksi, jopa useita kertoja päivässä. Koodin yhdistämisen yhteydessä voidaan automaattisesti ajaa regressiotestit ja siten saada välitöntä palautetta ohjelmiston toimivuudesta.



Kuva 2. Jatkuvan integraation osat. [Duvall *et al.* 2007]

Esimerkki jatkuvan integraation toiminnasta on esitetty kuvassa 2. Kehittäjä (developer) *puskee* (commit changes) lähdekoodimuutoksensa *versionhallintaan* (version control repository). Muutosten puskemisen jälkeen *jatkuvan integraation palvelin* (CI server, integration build machine) havaitsee muutosten tapahtuneen, jolloin se hakee uusimman version lähdekoodista ja ajaa *buildiskriptin* (build script). Buildiskriptillä tarkoitetaan skriptiä, jolla ohjelmisto voidaan rakentaa lähdekoodista ja sen jälkeen ajaa regressiotestit. Testit voidaan siis ajaa automaattisesti jatkuvan integraation palvelimella aina, kun versionhallintaan pusketaan muutoksia. Uudelleensuorittamalla regressiotestit joka muutoksen yhteydessä varmistetaan, että uusi muutos ohjelmistoon ei ole hajottanut jo olemassa olevaa toiminnallisuutta. Jatkuvan integraation palvelin voi lähettää testien tulokset kehittäjille esimerkiksi sähköpostilla (feedback mechanism). Jos jokin testi ei mene läpi, saavat kehittäjät siitä nopeaa palautetta ja ongelmaa ohjelmistossa voidaan alkaa korjaamaan heti. [Duvall *et al.* 2007]

Jatkuva integraatio voidaan myös yhdistää *jatkuvaan käyttöönottoon*. Kun jatkuvassa integraatiossa testataan ohjelmiston oikeellinen toiminta, voidaan jatkuvassa käyttöönotossa automatisoida myös ohjelmiston julkaisu. Jatkuva käyttöönotto voidaan liittää osaksi aiemmin mainittua buildiskriptiä.

Jenkins on automaatiopalvelin, jonka satojen laajennosten avulla voidaan tukea ohjelmistoprojektien rakentamista, käyttöönottoa ja automatisointia [Jenkins]. Jenkinsiä voidaan käyttää jatkuvan integraation palvelimena ja siihen voidaan myös sisällyttää jatkuvan käyttöönoton vaiheet. Jenkinsin käyttöä osana testausprosessia käsitellään myöhemmin kohdassa 5.3.

3. C#-testaus

John Deere Forestry Oy:llä C#-kehitysympäristönä käytetään Visual Studiota. Visual Studio on Microsoftin kehittämä ohjelmointiympäristö, josta löytyy esimerkiksi lähdekoodieditori, debuggeri, visuaalisia suunnittelutyökaluja ja paljon laajennoksia, jotka voivat mahdollistaa uutta toiminnallisuutta tai helpottaa kehitystyötä.

On olemassa monia .NET-ympäristön kanssa yhteensopivia testityökaluja, tässä luvussa käsitellään muutamaa niistä. Vertailuun on valittu kolme käytetyimmistä C#-yksikkötestikehyksistä, joille myös löytyy tuki Visual Studion Test Explorerista: MSTest, NUnit ja xUnit.net. Test Explorer -tuen ansiosta testejä pystyy ajamaan komentorivin lisäksi myös Visual Studion käyttöliittymästä [Microsoft 2019a]. Nämä kolme testikehystä valittiin Visual Studiosta löytyvän tuen lisäksi myös internetin keskustelupalstojen C#-yksikkötestaamiseen liittyvien keskustelujen ja erinäisten C#-yksikkötestikehysten vertailujen perusteella. Myös Khalid ja Naeem [2018] päätyivät omassa testityökalujen tutkimuksessaan vertailemaan MSTestiä, NUnitia ja xUnit.netiä toisiinsa.

Kaikki kolme vertailtavaa testikehystä kuuluvat xUnit-perheeseen. xUnit on kollektiivinen nimitys kaikille yksikkötestikehyksille, jotka pohjautuvat Kent Beckin SUnit-kehykseen ja siten käyttävät samoja toimintaperiaatteita kuin SUnit. SUnitin tärkeimpiin toimintaperiaatteisiin kuuluvat vakaa *testiympäristö* (test fixture), *testitapaus* (test case), *oikeellisuuden tarkistus* (check) ja *testisarja* (test suite) [Beck 1997].

Yksikkötestikehysten vertailun jälkeen käsitellään myös Visual Studion joidenkin versioiden sisältämää IntelliTest-testityökalua.

3.1. NUnit, MSTest ja xUnit.net

NUnit käännettiin alun perin JUnit-testikehyksen pohjalta toimimaan .NET-ympäristön testaamiseen samoin kuin JUnit toimii Java-ympäristössä. Vaikka tässä vertailussa keskitytään vain C#-kieleen, mainittakoon että NUnit tukee kaikkia .NET-ohjelmointikieliä.

MSTest on Microsoftin kehittämä testityökalu ohjelmistojen testaamiseen Visual Studiossa. MSTest sisältyy valmiiksi Visual Studioon, eikä vaadi liitännäisten asennusta toimiakseen. MSTestiä voi C#-kielen lisäksi käyttää myös F# ja Visual Basic -kielisten ohjelmistojen testaamiseen.

xUnit.net on ilmainen avoimen lähdekoodin yksikkötestityökalu .NET-kehykselle, jonka on kehittänyt NUnitin alkuperäinen kehittäjä. Kuten NUnit, myös xUnit.net tukee monia .NET-ohjelmointikieliä. [xUnit.net a]

xUnit.net kehitettiin modernimmaksi versioksi NUnitista. Brad Wilson sekä NUnitin alkuperäinen kehittäjä James Newkirk päättivät kehittää xUnit.netin modernimmaksi versioksi NUnitista, hyödyntäen myös uudempien .NET-kehyksen versioiden lisäämiä ominaisuuksia. [Newkirk 2007a]

3.2. Esimerkki C#-yksikkötestistä

Tässä kohdassa käydään läpi esimerkki yksikkötestistä, käyttäen MSTest-yksikkötestikehystä.

```
namespace Calculator
{
    public class Calculations
    {
        public int Add(int number1, int number2)
        {
            return number1 + number2;
        }
    }
}
```

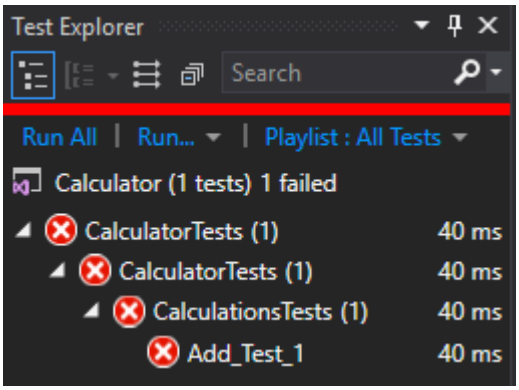
Koodikatkelma 1. Testattava luokka ja metodi C#-kielellä.

Koodikatkelmassa 1 on esitelty Visual Studiossa C#-kielellä kirjoitettu projekti, joka sisältää luokan nimeltä ”Calculations”. Calculations-luokka taas sisältää kahden kokonaisluvun yhteenlaskun suorittavan metodin nimeltä ”Add”, jota haluttaisiin nyt testata. Seuraavaksi täytyy luoda MSTest-testiprojekti, -luokka ja -metodeja.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Calculator;

namespace CalculatorTests
{
    [TestClass]
    public class CalculationsMSTestTests
    {
        [TestMethod]
        public void Add_Test_1()
        {
            // Arrange
            var calc = new Calculations();
            int number1 = 1;
            int number2 = 2;
            int expected = 4;

            // Act
            int actual = calc.Add(number1, number2);

            // Assert
            Assert.AreEqual(expected, actual);
        }
    }
}
```

Koodikatkelma 2. MSTest-testiluokka ja testimetodi.

Koodikatkelmassa 2 on MSTest-testiluokka ja testimetodi. Testimetodi Add_Test_1 on AAA-mallin mukaisesti jaettu kolmeen osaan: Arrange, Act ja Assert. Testimetodi testaa Add-metodia kutsumalla sitä kahdella pätevällä kokonaisluvulla.



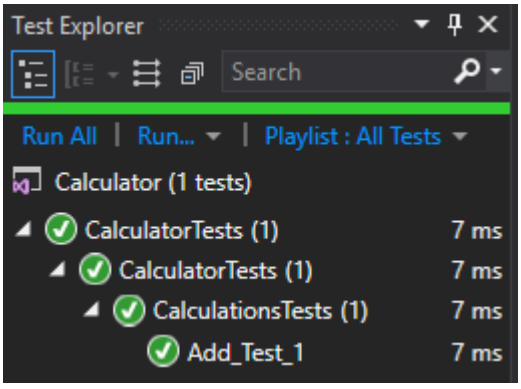
Kuva 3. Testitulokset, testi ei mennyt läpi.

Testi Add_Test_1 ei mennyt läpi, kuten kuvasta 3 näkee. Testimetodiin oli upotettu virhe, kuten koodikatkelmasta 2 saattaa huomata. Testi odotti Add-metodin tuloksen olevan neljä, vaikka se todellisuudessa on 1+2=3. Korjataan Add_Test_1 odottamaan laskun tulokseksi arvoa kolme.

```
[TestMethod]
public void Add_Test_1()
{
    // Arrange
    var calc = new Calculations();
    int number1 = 1;
    int number2 = 2;
    int expected = 3;

    // Act
    int actual = calc.Add(number1, number2);

    // Assert
    Assert.AreEqual(expected, actual);
}
```

Koodikatkelma 3. Korjattu testimetodi.

Nyt kun Add_Test_1-testimetodi on muutettu koodikatkelman 3 mukaiseksi, ajetaan testi uudestaan. Korjattu Add_Test_1-testimetodi meni läpi, kuten kuvassa 4 näkyy.



Kuva 4. Testitulokset, testi meni läpi.

3.3. Yksikkötestikehysten välisiä eroja

Näistä kolmesta yksikkötestikehyksestä vain MSTest tulee Visual Studion mukana, NUnit ja xUnit.net vaativat lisäliitännäisten asennusta Visual Studioon. xUnit.net ja NUnit siis vaativat enemmän asennustyötä lisäliitännäisten asennuksen muodossa, mutta niiden asennus on kuitenkin vain kertaluontoinen prosessi käytön alussa. Kerta-asennuksen jälkeen nekin toimivat pitkäaikaisesti MSTestin tavoin. NUnit ja xUnit.net voidaan asentaa Visual Studion sisältämän Microsoftin NuGet-pakettienhallintaohjelman kautta. Khalid ja Naeem [2018] argumentoivat MSTestin olevan helppokäyttöisempi kuin NUnit ja xUnit.net. MSTest, NUnit ja xUnit.net kaikki voivat ajaa testejä Visual Studion Test Explorerin kautta, suoraan Visual Studion käyttöliittymästä [Microsoft 2019a].

Selitys	MSTest v2	NUnit 3.x	xUnit.net 2.x
Testiluokka	[TestClass]	[TestFixture]	-
C#-projektin testien alustus	[AssemblyInitialize]	[SetUpFixture] + [OneTimeSetUp]	IDisposable luokan rakentaja
C#-projektin testien loppusiivous	[AssemblyCleanup]	[SetUpFixture] + [OneTimeTearDown]	IDisposable.Dispose
Testiluokan alustus	[ClassInitialize]	[OneTimeSetUp]	IClassFixture<T>
Testiluokan loppusiivous	[ClassCleanup]	[OneTimeTearDown]	IClassFixture<T>
Testimetodi	[TestMethod]	[Test]	[Fact]
Testin alustus	[TestInitialize]	[SetUp]	IDisposable luokan rakentaja
Testin loppusiivous	[TestCleanup]	[TearDown]	IDisposable.Dispose

Taulukko 3. Testien elinkaarimääritteet C#-yksikkötestikehyksissä.

Taulukossa 3 on esitetty testien elinkaarimääritteitä vertailtavilla testikehyksillä. Taulukosta voi huomata, että MSTestistä ja NUnitista löytyy attribuuttivastineet jokaiselle taulukossa mainitulle attribuutille, mutta xUnit.netistä ei. xUnit.netissä ei ole yhtä monia valmiita attribuutteja, mutta uusia attribuutteja voi määritellä itse xUnit.netin tarjoamien rajapintojen avulla. Myös NUnit 3 ja MSTest v2 tarjoavat mahdollisuuksia kehittää omia attribuutteja, mutta xUnit.netin valmiiden attribuuttien puute voi aiheuttaa päänvaivaa ainakin uusille käyttäjille.

3.4. NUnitista xUnit.netiin

Tässä kohdassa käydään läpi alkuperäisiä syitä xUnit.netin kehittämiselle. Samalla käy ilmi joitain eroja NUnitin, MSTestin ja xUnit.netin välillä.

3.4.1. Testien eristys

xUnit.netistä poistettiin NUnitin ja MSTestin tukemat testien alustus- ja purkumetodiattribuutit (NUnitissa SetUp ja TearDown, MSTestissä TestInitialize ja TestCleanup). Syitä vastaavien attribuuttien puuttumiselle xUnit.netissä on monia. James Newkirk [2007b] nosti esille muutaman kohtaamansa ongelman yhteisissä alustus- ja purkumetodeissa:

1. Jokaista testimetodia luettaessa täytyy katsoa myös alustus- ja purkumetodeja saadakseen täyden kontekstin testeihin. Tämä aiheuttaa ylimääräistä kelaamista metodista toiseen eli huonontaa testien luettavuutta.
2. Kaikkien testien jakamat testin alustus- ja purkumetodit tarkoittavat, että jokaisen testimetodin kohdalla tehtäisiin alustukset jokaiselle testille eikä vain kyseisen testin tarvitsemia alustuksia. Testeissä siis tehdään tarpeettomia alustuksia, jos jokainen testi ei vaadi juuri samoja alustustoimenpiteitä.
3. Kaikkien testien jakama alustusmetodi tarkoittaa, että testeissä joudutaan käyttämään yhteisiä jäsenmuuttujia, joka huonontaa testien eristystä toisistaan.

Näistä syistä Newkirk päätti jättää yksikkötestimetodien yhteiset alustus- ja purkumetodit pois xUnit.netistä. Yksikkötestin luettavuutta voitaisiin parantaa tekemällä vain testin tarvitsemat alustukset testin sisällä. Joissain testeissä voidaan vaatia samankaltaisia alustuksia, mutta jaettu alustusmetodi ei aina ole oikea ratkaisu.

"Some people will argue that you cannot build a tool that stops people from writing poor code. However, you can write a tool that prevents people from shooting themselves in the foot. [Newkirk 2007b]"

Tapaukset, joissa testit jakavat yhteisen olion ja muuttavat jaetun olion tilaa, voivat olla ongelmallisia. Testien tulisi olla eristettyjä toisistaan, jotta ne eivät vaikuttaisi toistensa tuloksiin. Jos testit muuttavat jaetun olion tilaa ilman, että oliota uudelleenalustetaan testien välissä, testien suoritusjärjestys saattaa vaikuttaa testien tuloksiin. Alustusten määrää saatetaan haluta minimoida testien suorituksen nopeuttamiseksi, mutta testit tulisi silti eristää toisistaan, jotta vältyttäisiin virheellisiltä tuloksilta. Jos testit on eristetty toisistaan, voi Visual Studiota käytettäessä ajaa testejä myös rinnakkaisesti, joka voi nopeuttaa testien ajoa huomattavasti.

3.4.2. Poikkeusten käsittely

xUnit.netistä puuttuu myös ExpectedException-attribuutti, joka löytyy MSTestistä ja NUnitista. Attribuutti kertoo testimetodin odottavan parametrina määritellyn poikkeuksen toteutumista, mutta ei ota kantaa siihen millä rivillä testimetodissa poikkeus esiintyy. Tämä attribuutti voi piilottaa oikeita virheitä, jos esimerkiksi testissä suoritetaan useampi metodikutsu ja poikkeuksen heittääkin jokin muu kutsuista kuin se, mistä poikkeusta odotettiin.

Poikkeuksen odottamista merkitsevän attribuutin sijasta xUnit.netissä poikkeuksen odottamiseen käytetään Assert.Throws-metodia, jonka avulla voidaan tarkentaa poikkeuksen tapahtumasijainti tiettyyn operaatioon testin sisällä, kuten esimerkiksi metodikutsuun [Newkirk 2007a]. Näin voidaan myös odottaa useaa erilaista poikkeusta eri osissa testiä. Vastaavat metodit on myöhemmin lisätty myös MSTestiin [Microsoft a] ja NUnitiin [NUnit 2019].

Ongelmallista ExpectedException-attribuutissa on myös, että se ei sovi yksikkötestien kirjoittamisessa käytettyyn AAA-malliin. ExpectedException-attribuutti ei sovi tähän malliin, koska attribuutin täytyy esiintyä koodissa juuri ennen testimetodia ja poikkeuksen odottaminen kuuluisi AAA-mallin mukaan testin Assert-lohkoon, eli sen loppuun.

3.4.3. Testien näkyvyys

MSTestissä testiluokkaa merkitään attribuutilla TestClass ja yksikkötestimetodien täytyy sijaita testiluokan sisällä. NUnitin versiosta 2.5 lähtien testiluokkaa merkitsevän TestFixture-attribuutin käytöstä tuli vapaaehtoista useimmissa tapauksissa [NUnit 2009]. xUnit.netissä ei käytetä lainkaan erillisiä testiluokkia vaan yksikkötestimetodit voivat sijaita missä vain julkisessa luokassa. Nykyisin MSTest on siis ainoa vertailun alla olevista yksikkötestikehyksistä, jossa testiluokka tarvitsee merkitä erikseen attribuutilla. Vaikka testeille ei tarvitsisikaan erillistä attribuutilla merkittyä testiluokkaa, testien sijoittaminen omaan luokkaansa voi silti olla hyvä idea testimetodien eristämiseksi muusta toteutusta sisältävästä lähdekoodista.

3.5. Parametrisoidut yksikkötestit

Parametrisoidussa yksikkötestissä testille annetaan parametrina joitain arvoja. Parametrisoinnin avulla testi voidaan ajaa useita kertoja eri syötearvoilla. Tässä kohdassa käydään läpi esimerkkejä parametrisoiduista testeistä ja vertaillaan testikehyksien eroja liittyen testien parametrisointiin.

3.5.1. Staattiset ja dynaamiset parametrit

Koodikatkelmassa 4 on DataRow-attribuutin avulla staattisilla parametreilla parametrisoitu yksikkötesti MSTest-testikehyksessä, jossa testataan aiemmin koodikatkelmassa 1 esiteltyä Add-metodia.

```
[TestMethod]
[DataRow(1, 2, 3)]
[DataRow(1, -1, 0)]
[DataRow(-2, -7, -9)]
[DataRow(int.MinValue, -1, int.MaxValue)]
public void Add_Test_With_DataRow(int number1, int number2, int expected)
{
    // Arrange
    var calc = new Calculations();

    // Act
    var result = calc.Add(number1, number2);

    // Assert
    Assert.AreEqual(expected, result);
}
```

Koodikatkelma 4. Staattisilla parametreilla parametrisoitu testi MSTest-testikehyksessä.

Yksi DataRow-attribuutti sisältää testimetodin parametrien arvot yhdelle testiajolle. Koodikatkelman 4 testimetodi Add_Test_With_DataRow ajetaan neljä kertaa DataRow-attribuuteissa määrätyillä syötteillä. Testiajon parametrit määritellään DataRow-attribuutissa samassa järjestyksessä kuin parametrit on esitelty testimetodissa. Kaikki neljä testiajoa läpäisivät testin.

```
[TestMethod]
[DynamicData("GetSourceData", DynamicDataSourceType.Method)]
public void Add_Test_With_DynamicData(int number1, int number2, int expected)
{
    // Arrange
    var calc = new Calculations();

    // Act
    var result = calc.Add(number1, number2);

    // Assert
    Assert.AreEqual(expected, result);
}

public static IEnumerable<object[]> GetSourceData()
{
    var allData = new List<object[]> { };
    for (int i = 0; i < 10; i++)
    {
        allData.Add(new object[] {i, -i, 0});
    }

    return allData;
}
```

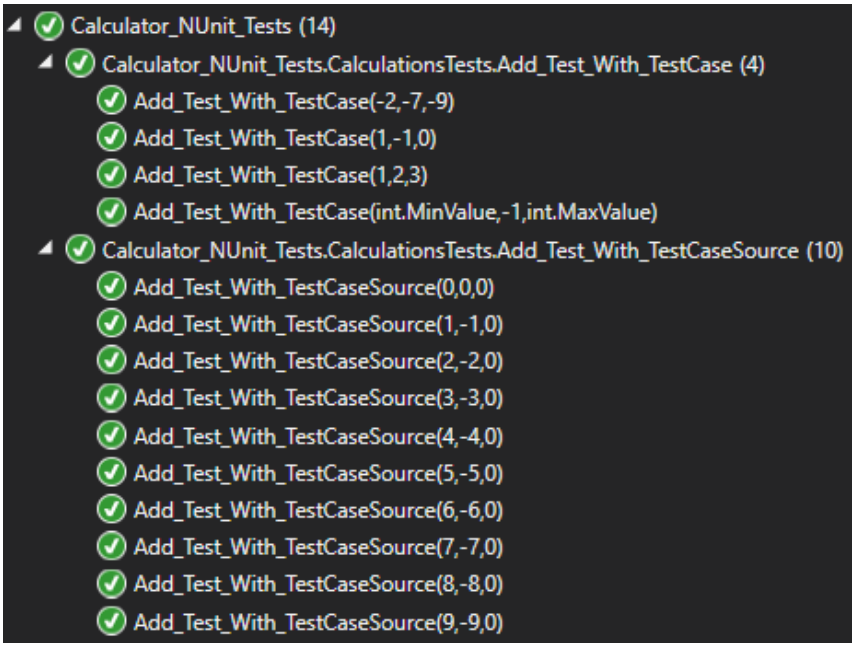
Koodikatkelma 5. Dynaamisilla parametreilla parametrisoitu testi MSTest-testikehyksessä.

Koodikatkelmassa 5 on dynaamisilla parametreilla parametrisoitu yksikkötesti Add_Test_With_DynamicData. Tämän testin parametrit luodaan metodissa GetSourceData dynaamisesti listaksi, jossa yksi listaelementti sisältää yhden testiajon parametrit,

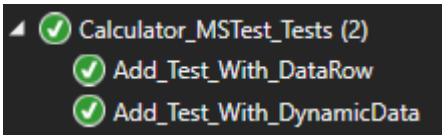
kuten koodikatkelman 4 yhdessä DataRow-attribuutissa. Testimetodin attribuutilla DynamicData testimetodiin saadaan liitettyä dynaamisia parametreja. Testiajojen parametrit voi toki esitellä myös DataRow-attribuuteissa, mutta mitä enemmän testiajoja eri parametreilla halutaan tehdä, sitä tärkeämpää dynaaminen parametrien generointi on. Jos yhdelle testimetodille halutaan tehdä esimerkiksi satoja tai jopa tuhansia testiajoja eri parametreilla, parametrien yksitellen esitteleminen DataRow-attribuuteilla voi olla melko epäkäytännöllistä.

Koodikatkelmassa 5 parametrilistaan lisätään arvoja kymmenelle testiajolle. Ensimmäinen Add-metodissa yhteenlaskettavista numeroista saa arvon väliltä 0−9 ja toinen numero vastaavasti väliltä 0−(-9). Odotettu Add-metodin tulos on kaikissa kymmenessä testiajossa nolla. Kaikki testiajot menivät läpi, sillä 0+(-0)=0, 1+(-1)=0, 2+(-2)=0 ja niin edelleen.

MSTestiä, NUnitia ja xUnit.netiä voi kaikkia käyttää Visual Studion Test Explorerissa, mutta niissä on kuitenkin joitain parametrisointiin liittyviä eroja Test Explorerin sisällä.



Kuva 5. Parametrillisten NUnit-testien tulokset Visual Studion Test Explorerissa.



Kuva 6. Parametrillisten MSTest-testien tulokset Visual Studion Test Explorerissa.

Test Explorer näyttää NUnit- ja xUnit.net-testeillä testien jokaiselle ajolle oman tulosrivinsä, kuten kuvassa 5. MSTest-testeille näkyy vain yksi tulosrivi testiä kohden, kuten kuvassa 6, vaikka sama testi olisi ajettu useamman kerran eri syötteillä. MSTestiä

käytettäessä yksittäisten testiajojen tulokset löytyvät vasta Test Explorerin testitulosten lisätietopaneelista.

3.5.2. Kombinatoriset parametrit

Taulukossa 4 on esitelty testiparametreihin liittyviä attribuutteja eri yksikkötestikehyksissä. MSTest ja xUnit.net eivät tue kombinatorisia, peräkkäisiä tai parittaisia parametreja valmiiden attribuuttien avulla.

Selitys	MSTest v2	NUnit 3.x	xUnit.net 2.x
Staattiset parametrit	[DataRow]	[TestCase]	[InlineData]
Dynaamiset parametrit	[DynamicData]	[TestCaseSource]	[MemberData]
Kombinatoriset parametrit	-	[Combinatorial]	-
Peräkkäiset parametrit	-	[Sequential]	-
Parittaiset parametrit	-	[Pairwise]	-

Taulukko 4. Testiparametreihin liittyviä attribuutteja eri yksikkötestikehyksissä.

```
[Test, Combinatorial]
public void Parameterized_Test(
    [Values("a string", "another", "one more")] string a,
    [Values(3, -2, 0)] int b,
    [Values(true, false)] bool c) { ... }
```

Koodikatkelma 6. Kombinatorisia parametreja NUnit-testikehyksessä.

Koodikatkelmassa 6 on esimerkki *kombinatorisista parametreista*. Kombinatorisilla parametreilla varustettu testi ajetaan kaikilla sen parametrien arvojen mahdollisilla yhdistelmillä. NUnit-kehyksessä kombinatorisia parametreja voi luoda käyttäen attribuuttia Combinatorial. Koodikatkelman 6 testi ajetaan 3*3*2=18 kertaa, sillä parametrille ”a” on määritelty kolme mahdollista arvoa, parametrille ”b” kolme mahdollista arvoa ja parametrille ”c” kaksi mahdollista arvoa. Kaikki eri parametrien yhdistelmät ajetaan omana testiajonaan. Koodikatkelmassa 7 on havainnollistava esimerkki koodikatkelman 6 testin ensimmäisten kahdeksan ajon parametriyhdistelmistä.

```
Parameterized_Test("a string", 3, true)
Parameterized_Test("a string", 3, false)
Parameterized_Test("a string", -2, true)
Parameterized_Test("a string", -2, false)
Parameterized_Test("a string", 0, true)
Parameterized_Test("a string", 0, false)
Parameterized_Test("another", 3, true)
```

```
Parameterized_Test("another", 3, false)
...
```

Koodikatkelma 7. Kombinatoristen parametrien testin esimerkkiajoja.

Vaikka MSTest ja xUnit.net eivät tuekaan Combinatorial-attribuuttia tai vastaavaa, sama lopputulos voidaan näissä kehyksissä saavuttaa myös esimerkiksi dynaamisten parametrien attribuuttien avulla, vaikkakaan ei yhtä siististi tai helposti. NUnit-kehys sisältää myös attribuutit Sequential ja Pairwise, joilla voidaan Combinatorial-attribuutin kaltaisesti generoida testiajoja parametreille annettujen arvojen pohjalta.

```
Parameterized_Test("a string", 3, true)
Parameterized_Test("another", -2, false)
Parameterized_Test("one more", 0, null)
```

Koodikatkelma 8. Peräkkäisten parametrien testin esimerkkiajoja.

Jos koodikatkelman 6 metodi merkittäisiin Combinatorial-attribuutin sijasta Sequential-attribuutilla, kyseessä olisi *peräkkäiset parametrit*, jotka tuottaisivat koodikatkelman 8 mukaiset testiajot. Sequential-attribuutilla varustettu NUnit-testimetodi ajaa testin ensin kaikkien parametrien ensimmäisillä arvoilla, sitten toisilla arvoilla jne. Ajoja suoritetaan niin monta, kuin parametreissa on enimmillään arvoja. Puuttuvat arvot korvataan null-arvoilla, kuten koodikatkelman 8 kolmannen ajon parametri ”c”.

NUnit-kehyksen Pairwise-attribuutilla luodaan *parittaisia parametreja*, joilla pyritään rajoittamaan Combinatorial-attribuutin luomien testiajojen määrän eksponentiaalista kasvua parametrien määrän ja niiden arvojen määrän lisääntyessä. Pairwise-attribuutilla merkitty testimetodi suorittaa tarvittavan monta testiajoa, jotta testi on ajettu kaikilla mahdollisilla parametrien arvopareilla.

```
Parameterized_Test("a string", 0, false)
Parameterized_Test("a string", -2, true)
Parameterized_Test("a string", 3, true)
Parameterized_Test("another", 0, true)
Parameterized_Test("another", -2, false)
Parameterized_Test("another", 3, false)
Parameterized_Test("one more", 0, true)
Parameterized_Test("one more", -2, true)
Parameterized_Test("one more", 3, false)
```

Koodikatkelma 9. Parittaisten parametrien testin esimerkkiajoja.

Koodikatkelmassa 9 on koodikatkelman 6 testimetodin ajamat testiajot, jos testimetodin Combinatorial-attribuutin tilalla olisi Pairwise-attribuutti. Pairwise tuottaa vähemmän testiajoja kuin Combinatorial, mutta kuitenkin jokaisen parametriparin jokaisen arvoyhdistelmän. Tässä testimetodissa Combinatorial-attribuutti tuotti 18 testiajoa, kun taas Pairwise-attribuutti vähensi ajojen määrän 9 testiajoon.

3.6. Testien kategorisointi

Testejä voidaan *kategorisoida*, jotta voitaisiin ajaa vain osa testeistä (tietyt kategoriat) kerrallaan kaikkien testien sijasta. Kerralla ajettavien testien rajoittaminen voi olla hyödyllistä, jos testejä on paljon tai jos kaikkien testien suorittamisessa kuluisi huomattavasti aikaa. Taulukossa 5 on esitetty yksikkötestikehysten sisältämiä testien kategorisointiin liittyviä attribuutteja.

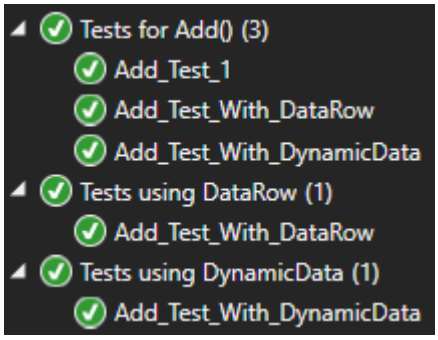
Selitys	MSTest v2	NUnit 3.x	xUnit.net 2.x
Testin kategorisointi	[TestCategory(<kategoria>)]	[Category(<kategoria>)]	[Trait("Category", <kategoria>)]
Metadatan lisäys testiin	[TestProperty(<tyyppi>,<arvo>)]	[Property(<tyyppi>,<arvo>)]	[Trait(<tyyppi>, <arvo>)]
Testin prioriteetti	[Priority(<arvo>)]	[Property(”Priority”,<arvo>)]	[Trait(”Priority”, <arvo>)]
Testin ohitus	[Ignore(<syy>)]	[Ignore(<syy>)]	[Fact(Skip=<syy>)]

Taulukko 5. Testien kategorisointiin liittyviä attribuutteja C#-yksikkötestikehyksissä.

MSTestissä ja NUnitissa testien kategorisointiin voidaan käyttää sille dedikoitua attribuuttia, MSTestissä TestCategory ja NUnitissa Category, joille annetaan kategorian nimi parametrina. *Metadataa* testeille voidaan lisätä MSTestin TestProperty, NUnitin Property ja xUnit.netin Trait -attribuuteilla. Koska xUnit.netissä ei ole dedikoitua attribuuttia kategorioille, käytetään kategorisointiin Trait-attribuuttia. Kaikissa testikehyksissä kategoriat ovat teknisesti vain yhden tyyppistä metadataa, mutta MSTestissä ja NUnitissa kategorioiden asettamiselle on kuitenkin kehitetty omat attribuutit. Testimetodin kategorisointi MSTestissä onnistuu koodikatkelman 10 mukaisesti.

```
[TestMethod]
[TestCategory("Tests for Add()")]
public void Add_Test_1() { ... }
```

Koodikatkelma 10. Testimetodin kategorisointi MSTestissä.



Kuva 7. MSTest-testimetodien kategorisointia Visual Studion Test Explorerissa.

Testimetodit voi järjestää kategorioittain Visual Studion Test Explorerissa. Kuvassa 7 on kolme testimetodia jaettuna kolmeen kategoriaan. Testimetodille voi asettaa useita kategorioita tai muuta metadataa kaikissa kolmessa testikehyksessä. TestProperty-, Property- ja Trait-attribuuteilla voidaan lisätä testimetodiin metadataa, kuten vaikka testin kirjoittajan nimi. Koodikatkelmassa 11 on esimerkki metadatan lisäämisestä MSTest-testikehyksen testimetodille.

```
[TestMethod]
[TestProperty("Author","Teisko")]
public void Add_Test_1() { ... }
```

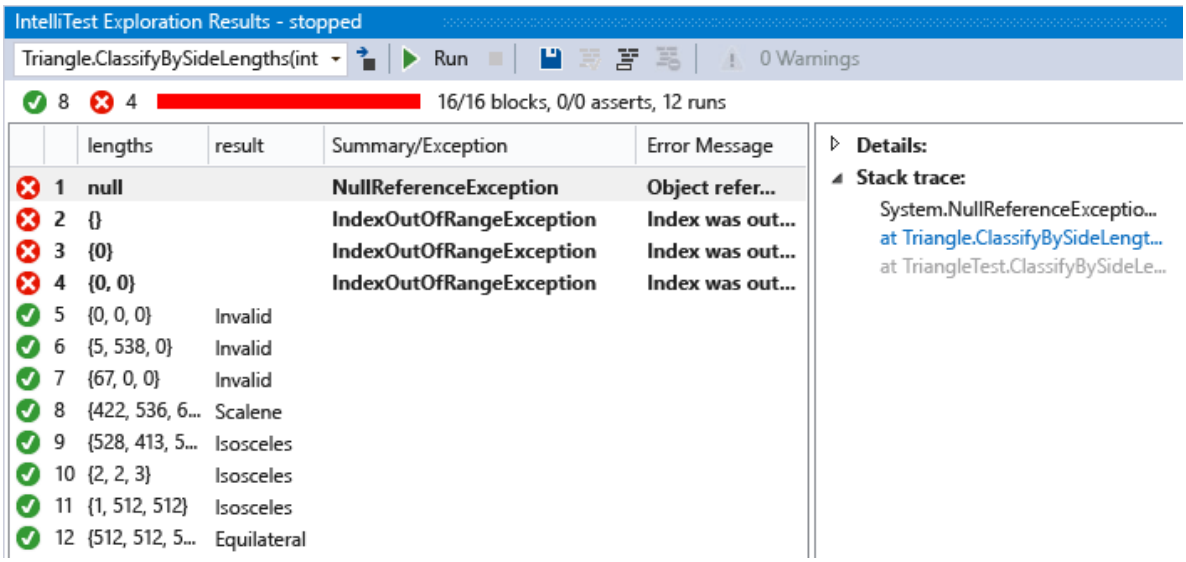
Koodikatkelma 11. Metadatan lisääminen testimetodille MSTestissä.

Testeille voidaan asettaa *prioriteettiarvoja* kaikissa kolmessa testikehyksessä. Testeille voidaan kaikissa testikehyksissä myös asettaa *ohitusattribuutti*, jonka avulla testiä ei koskaan ajeta. Ohitusattribuutit ovat hyödyllisiä esimerkiksi keskeneräisten testimetodien kanssa, kun tiedetään että testi ei vielä toimi halutulla tavalla, joten sitä ei haluta ajaa missään tilanteessa. Kun testi on valmis ajettavaksi, ohitusattribuutin voi poistaa, jolloin testi on taas ajettavissa. Kaikkia kategoria-, metadata- ja prioriteettiattribuutteja voidaan käyttää suodattimina ajettavien testien valinnassa, jos ei aina haluta ajaa jokaista testiä kerralla.

Yksi käyttökohde testien kategorisoinnille on integraatiotestit. Yksikkötestit ja integraatiotestit kannattaa pitää erillään toisistaan, esimerkiksi erillisissä testiprojekteissa tai kategorisoituina yksikkötesteiksi ja integraatiotesteiksi. Erityyppisten testien erottaminen toisistaan on tärkeää, sillä erotuksen avulla voidaan määritellä ajettavaksi joko vain yksikkötestit tai vain integraatiotestit. Yksikkötestit antavat kehittäjälle nopean palautteen yksikön toiminnasta ja integraatiotestit ovat yleisesti hitaampia suorittaa kuin yksikkötestit. Jos integraatiotestien suorittamisessa kestää huomattavasti pidempään kuin yksikkötesteissä, niitä ei välttämättä haluta ajaa joka tilanteessa nopeampien yksikkötestien kanssa.

3.7. IntelliTest

Visual Studio Enterprise Edition 2015 ja 2017 sisältävät myös *IntelliTest*-työkalun, joka analysoi C#-metodin logiikkaa ja sen kaikki suorituspolut. Analyysin pohjalta IntelliTest generoi metodille yksikkötestejä korkealla koodin kattavuudella [Microsoft 2015]. Generoidut yksikkötestit siis testaavat lähes kaikki suorituspolut metodin läpi. IntelliTest tallentaa generoimansa yksikkötestit uuteen tiedostoon, josta niitä voi ajaa myöhemmin uudestaan. MSTest ja NUnit tukevat IntelliTestiä [Microsoft 2015], mutta xUnit.netistä ei ainakaan toistaiseksi löydy IntelliTest-tukea [Tsai 2019].



Kuva 8. IntelliTestin generoimia testejä yhdelle metodille. [Microsoft 2015]

IntelliTest näyttää mitkä sen generoimista yksikkötesteistä kaatuvat, mitkä menevät läpi ja minkälaisen tuloksen testiajot antoivat. Kuvan 8 testit kaatuivat aina, jos metodin parametrina saadun kokonaislukutaulukon pituus oli alle kolme elementtiä. IntelliTest löytää syötteet, joilla metodin saa kaatumaan. Läpäistyjen testien osalta testaajan tulisi kuitenkin tarkistaa generoitujen testien tuloksien oikeellisuus. Vaikka testiajo ei kaatuisikaan, IntelliTest ei osaa automaattisesti päätellä ovatko testien tulokset toivotun mukaisia vaiko eivät [Microsoft 2015]. Läpäistyjen testien tulosten oikeellisuuden voi tarkistaa manuaalisesti IntelliTestin testituloksista. Testien tulosten oikeellisuuden tarkistamiseksi automaattisesti testeihin täytyy kuitenkin itse lisätä tulosten tarkistus, jos tuloksia halutaan automaattisesti tarkistaa laajemmin kuin ”kaatuiko vai eikö” -tasolla.

4. Angular-testaus

Angular on Googlen kehittämä avoimen lähdekoodin verkkosovelluskehys. Angularista käytettiin ennen nimitystä AngularJS, mutta versiosta 2.0.0 lähtien nimitys muutettiin vain Angulariksi [Fluin 2017]. Angular sisältää kirjastoja näkymäkomponenttien luomiselle (components), backendin kanssa kommunikoimiselle (@angular/common/http) ja näkymien reititykselle (@angular/router).

Angular pohjautuu Microsoftin kehittämään TypeScript-ohjelmointikieleen. TypeScript on JavaScriptin ylijoukko, joka kääntyy JavaScriptiksi. TypeScript tarjoaa lisäominaisuuksia tavalliseen JavaScriptiin nähden, kuten valinnaisen käännöksenaikaisen staattisen tyypityksen. Vahva tyypitys mahdollistaa tyyppien epäsopivuuksien tunnistamisen jo kääntämisen aikana. Heikolla tyypityksellä tällaiset virheet voitaisiin löytää vasta ajon aikana virheen tapahtuessa.

Heikon tyypityksen lisäksi JavaScript-sovellusten testaamiseen liittyy muitakin mahdollisia sudenkuoppia, joita monessa muussa kielessä ei ole. Sovellukset täytyy testata monella eri verkkoselaimella, sillä JavaScript-koodi ei välttämättä käyttäydy samalla tavalla eri verkkoselaimissa. Yhteensopivuusongelmien välttämiseksi sovelluksia tulee testata monilla eri verkkoselaimilla. Angularin testaamisessa käytetään neljän tyyppisiä työkaluja:

1. *Testikehys*: sisältää testien kirjoittamiseen vaaditut rajapinnat.
2. *Assertiokirjasto*: kirjasto, joka sisältää metodeja testin tuloksien oikeellisuuden tarkistamiseen. Testikehykset yleensä sisältävät jonkin assertiokirjaston.
3. *Yksikkötestiajuri*: yksikkötestien ajamisen automatisointiin joko verkkoselaimessa tai ilman verkkoselainta.
4. *End-to-end-testiajuri*: valmiin ohjelmiston end-to-end-testien ajamiseen verkkoselaimessa.

Oletuksena Angular CLI:n asennus sisältää testikehys Jasminen, yksikkötestiajuri Karman ja end-to-end-testiajuri Protractorin. Kaikki nämä kolme työkalua kuitenkin voidaan myös vaihtaa muihin vastaavia toimintoja tarjoaviin vaihtoehtoihin. Tässä luvussa käsitellään eri testikehyksiä, assertiokirjastoja, yksikkötestiajureita ja end-to-end-testiajureita, joita voidaan käyttää Angularin testaamiseen.

4.1. Angular-testikehykset

Angularin testaamiseen on saatavilla monia eri testikehyksiä. The State of JavaScript 2019 -kyselyn perusteella vuoden 2019 tunnetuimmat kolme JavaScript-testikehystä olivat Mocha, Jest ja Jasmine [Greif and Benitte 2019]. Myös monet verkosta löytyvät JavaScript-testikehysvertailut kertovat samaa tarinaa, joten nämä kolme testikehystä valittiin vertailtavaksi tässä työssä: Jasmine, Jest ja Mocha.

Jasmine on käyttäytymisvetoinen testikehys JavaScript-koodin testaamiseen. Jasmine pyrkii sisältämään kaiken testaukseen tarvittavan yhdessä paketissa. Jasmine on Angularin oletusarvoinen testikehys, sillä se sisältyy Angular CLI:n asennukseen. Angular myös tarjoaa ohjeita Angularin ja Jasminen yhteiskäyttöön [Angular]. Jasminea voi käyttää testaamiseen ilman testiajuria, mutta testiajurin käyttö helpottaa testien ajoa. Jasminen kanssa käytettäviksi testiajureiksi suositellaan yksikkötestiajuri Karmaa ja end-to-end-testiajuri Protractoria, nämä myös sisältyvät Angular CLI:n asennukseen.

Jest on Facebookin kehittämä testikehys JavaScript-koodin testaamiseen. Jestiä suositellaan React-sovellusten testaamiseen, mutta sitä voi käyttää myös Angular-sovellusten testaamiseen. Jest pohjautuu Jasmineen, jonka vuoksi Jest myös jakaa Jasminen testisyntaksin. Jestin ja Jasminen eroja käsitellään myöhemmin kohdassa 5.2.

Mocha on alun perin Node.js-palvelinsovelluksia varten kehitetty testikehys, mutta se tukee myös selaintestaamista. Mocha ei sisällä omaa assertiokirjastoa, mutta testeissä voidaan käyttää Node.js:n sisältämää Assert-moduulia [Node.js]. Riippuen kirjoitettavista testeistä, Assert-moduuli ei välttämättä ole tarpeeksi monipuolinen. Tällaisissa tapauksissa Mochan kanssa voidaan käyttää jotain muuta erillistä assertiokirjastoa, jossa on monipuolisemmat ominaisuudet.

4.2. Assertiokirjastot

Jasmine ja Jest sisältävät omat assertiokirjastonsa, joten erillistä assertiokirjastoa ei välttämättä tarvita. Myös Mochan kanssa voidaan käyttää Node.js:n sisältämää Assert-moduulia, joten Mochankaan kanssa ei välttämättä tarvita erillistä assertiokirjastoa. On kuitenkin hyvä tiedostaa, että vaihtoehtojakin löytyy.

Käytettävän assertiokirjaston valintaan vaikuttaa sen ominaisuuksien lisäksi myös kirjaston käyttämä assertiotyyli, kuten ”assert”, ”expect” tai ”should”. Koodikatkelmassa 12 on esimerkit näistä kolmesta assertiotyylistä. Kaikki nämä tyylit johtavat samaan tulokseen, mutta kehittäjillä voi olla erilaisia mieltymyksiä testien luettavuuden suhteen.

```
var foo = 'bar';

// Assert style
assert.equal(foo, 'bar');

// Expect style
expect(foo).toEqual('bar');

// Should style
foo.should.equal('bar');
```

Koodikatkelma 12. Erilaisia assertiotyylejä.

Jos Assert-moduulia ei haluta käyttää, Mocha myös ehdottaa viittä eri vaihtoehtoa Mochan kanssa käytettäväksi ulkoiseksi assertiokirjastoksi [Mocha 2020]:

1. Better-assert
2. Should.js
3. Expect.js
4. Unexpected
5. Chai

Näitä assertiokirjastoja voidaan käyttää myös muidenkin testikehysten kuin Mochan kanssa. Seuraavaksi käydään läpi pikaiset katsaukset Assert-moduulista ja näistä viidestä assertiokirjastosta. Assertiokirjastojen syntakseja havainnollistetaan kirjoittamalla esimerkkitestejä niillä.

4.2.1. Assert

Assert on yksinkertaisin tapa toteuttaa yksikkötestejä Node.js-sovelluksille. Node.js sisältää Assert-moduulin valmiiksi, joten sitä ei tarvitse asentaa erikseen. Assert ei kuitenkaan tarjoa yhtä monipuolisia ominaisuuksia kuin jotkin muut assertiokirjastot. Assert-moduuli mainitaan ensimmäistä kertaa jo Node.js:n version 0.1.22 dokumentaatiossa [Dahl 2009], joten se on ollut osa Node.js:ää lähes alusta asti. Vaikka Assert onkin ollut osa Node.js:ää jo pitkään ja siihen on lisätty ominaisuuksia, se on silti pyritty pitämään mahdollisimman minimaalisena kokonaisuutena. Assert-moduuliin ei haluttu lisätä monia uusia ominaisuuksia, vaan käyttäjiä kehotettiin mieluummin käyttämään muita assertiokirjastoja tai luomaan omia kirjastojaan, jotka toteuttaisivat ehdotettuja uusia ominaisuuksia [GitHub 2012a; GitHub 2012b; GitHub 2013].

Koodikatkelmassa 13 on esimerkki Assert-moduulilla kirjoitettujen testien syntaksista. Testeillä testataan käyttäjää, jolla on nimi ja lemmikkejä. Testeissä testataan käyttäjän nimen olemassaoloa, sen tietotyyppiä, pituutta ja arvoa. Lisäksi testataan lemmikkien olemassaoloa ja niiden määrää.

```
var assert = require('assert');

var user = {
  name: 'John',
  pets: ['Poppy', 'Bella', 'Charlie', 'Daisy', 'Alfie']
};

assert('name' in user);
assert(typeof user.name === 'string');
assert(user.name.length == 4);
assert.equal(user.name, 'John');
assert('pets' in user);
assert(user.pets.length, 5);
```

Koodikatkelma 13. Esimerkki Assert-assertiokirjaston käytöstä.

Vuonna 2016 Stack Overflow -sivustolla kysyttiin miksi Node.js varoittaa Assert-moduulin käytöstä yleiskäyttöisenä testikehyksenä. Kysymykseen vastasi Rich Trott, yksi Node.js:n kehittäjistä. Syynä oli Assert-moduulissa piilevät rajatapausbugit ja siitä puuttuvat ominaisuudet. Trott myös kertoo, että jos Node.js:n kehitys aloitettaisiin alusta, ei Assert-moduulia välttämättä luotaisi siihen lainkaan. Syynä tälle ovat monet muut assertiokirjastot, jotka ovat parempia kuin Node.js:n Assert-moduuli, ja joita päivitetään nopeammalla tahdilla. Assert-moduuli siis sopi yksinkertaisiin testeihin, mutta siinä se. Huhtikuusta 2017 lähtien Node.js ei kuitenkaan ole enää varoittanut Assert-moduulin käytöstä, joten Trottin esille nostamat argumentit voivat nyt olla jo vanhentuneita. [Stack Overflow 2017]

4.2.2. Better-assert

Better-assert on assertiokirjasto, joka lupaa parempia ”assert”-tyylisiä assertioita Node.js-sovelluksille, kuin Assert-moduuli tarjoaa. Better-assertin virheviestit näyttävät pinojäljityksen, virheen aiheuttaneen lauseen ja lauseen rivinumeron. Nämä ominaisuudet eivät kuitenkaan vaikuta kovin houkuttelevilta syiltä käyttää Better-assertia, sillä nykyään myös Node.js:n valmiiksi sisältämä Assert-moduuli näyttäisi tarjoavan kaikki samat ominaisuudet. Nämä ominaisuudet eivät kuitenkaan aina ole olleet Assert-moduulissa, sillä Assert-moduuli haluttiin pitää mahdollisimman yksinkertaisena. Myös Assert-moduulin virheviesteihin on sittemmin kuitenkin lisätty pinojäljitykset, virheen aiheuttanut lause ja lauseen rivinumero. Kaikki käyttäjät eivät kuitenkaan ole olleet tyytyväisiä Assert-moduulin kehityksen jatkumiseen, kun tarjolla on muitakin vaihtoehtoja assertiokirjastoksi ja kehitykseen kuluvat resurssit voitaisiin käyttää toisaalle [GitHub 2018a].

Koodikatkelmassa 14 on esimerkki Better-assertin syntaksista. Testeissä testataan samoja asioita, kuin edellisessä Assert-moduulin esimerkissä. Myös syntaksi Assert-moduulin ja Better-assert-kirjaston välillä on melkein sama, sillä molemmat kirjastot edustavat ”assert”-assertiotyyliä.

```
var betterAssert = require('better-assert');

var user = {
  name: 'John',
  pets: ['Poppy', 'Bella', 'Charlie', 'Daisy', 'Alfie']
};

betterAssert('name' in user);
betterAssert(typeof user.name === 'string');
betterAssert(user.name.length == 4);
betterAssert(user.name == 'John');
betterAssert('pets' in user);
betterAssert(user.pets.length, 5);
```

Koodikatkelma 14. Esimerkki Better-assert-assertiokirjaston käytöstä.

4.2.3. Should.js

Should.js on kuvaava, helposti luettava ja testikehysagnostinen assertiokirjasto. Koodikatkelmassa 12 esitetyistä assertiotyyleistä Should.js edustaa ”should”-tyyliä. Should.js:n syntaksi tukee käyttäytymisvetoista testaamista, muistuttaa luonnollista kieltä ja on siten myös helppolukuista. Luonnollisen kielen sanat on erotettu toisistaan pistenotaatiolla, kuten koodikatkelman 15 esimerkkitesteistä käy ilmi. Kuten esimerkin lopussa on esitetty, Should.js:n assertioita voi käyttää joko olioiden kutsuttavana metodina tai erillisenä funktiona, jolle annetaan testattava arvo parametrina. Should.js toimii jatkeena Object.prototype-oliolle, joten should-metodia pystyy kutsumaan lähes kaikista JavaScript-olioista. Should-assertioihin ketjutetaan muita assertiometodeja samalla tavalla, riippumatta käyttääkö olion kutsuttavaa metodia vai erillistä should-funktiota.

```
var should = require('should');

var user = {
  name: 'John',
  pets: ['Poppy', 'Bella', 'Charlie', 'Daisy', 'Alfie']
};

user.should.have.property('name').which.is.a.String();
user.name.should.have.lengthOf(4);
user.name.should.equal('John');
user.should.have.property('pets').with.lengthOf(5);

// Results of <object>.should and should(<object>) in most situations are the
// same
user.should.have.property('name', 'John');
should(user).have.property('name', 'John');
```

Koodikatkelma 15. Esimerkki Should.js-assertiokirjaston käytöstä.

4.2.4. Expect.js

Expect.js on käyttäytymisvetoinen assertiokirjasto, joka pohjautuu vahvasti Should.js-kirjastoon. Toisin kuin Should.js, Expect.js käyttää ”expect”-tyylisiä assertioita. Assertiot ovat silti käyttäytymisvetoisia ja ne käyttävät luonnollista kieltä ja pistenotaatiota. Koodikatkelmassa 16 on esimerkki Expect.js:n syntaksista. Esimerkissä testataan samat asiat kuin aiemmissakin testeissä. Expect.js:n syntaksi muistuttaa erityisen läheisesti Should.js:n syntaksia erillisenä funktiona käytettäessä.

```
var expect = require('expect.js');

var user = {
  name: 'John',
  pets: ['Poppy', 'Bella', 'Charlie', 'Daisy', 'Alfie']
};
```

```
expect(user).to.have.property('name');
expect(user.name).to.be.a('string').length(4);
expect(user.name).to.equal('John');
expect(user).to.have.property('pets').length(5);
```

Koodikatkelma 16. Esimerkki Expect.js-assertiokirjaston käytöstä.

4.2.5. Unexpected

Unexpected-assertiokirjasto käyttää myös ”expect”-tyylisiä assertioita, mutta kuitenkin erilaisella syntaksilla kuin Expect.js. Expect.js:ssä luonnollista kieltä ketjutetaan piste-notaation avulla, kun taas Unexpected:ssä luonnollinen kieli ilmaistaan enimmäkseen string-parametreina. Lauseiden luettavuus pysyy jotakuinkin samana, mutta erinäköisellä syntaksilla. Koodikatkelmassa 17 on esimerkki Unexpectedin syntaksista.

```
var expect = require('unexpected');

var user = {
  name: 'John',
  pets: ['Poppy', 'Bella', 'Charlie', 'Daisy', 'Alfie']
};

expect(user, 'to have property', 'name');
expect(user.name, 'to be a', 'string').and('to have length', 4);
expect(user.name, 'to equal', 'John');
expect(user, 'to have property', 'pets');
expect(user.pets, 'to have length', 5);
```

Koodikatkelma 17. Esimerkki Unexpected-assertiokirjaston käytöstä.

Unexpectedin luoja Sune Simonsen käytti ennen Expect.js-assertiokirjastoa, mutta totesi Expect.js:n olevan turhan hidas. Expect.js:n hitaus antoi Simonsenille idean yrittää kehittää uuden tehokkaamman assertiokirjaston. Assertioiden ilmaiseminen merkkijonoina Unexpectedissä Expect.js:n käyttämien metodiketjujen sijasta mahdollisti suorituskyvyn huomattavan paranemisen. [Simonsen 2016]

4.2.6. Chai

Chai on assertiokirjasto, joka tukee kaikkia kolmea edellä mainituista assertiotyyleistä: ”should”, ”expect” ja ”assert”. Chain assert-tyyli muistuttaa Node.js:n Assert-moduulin syntaksia, should-tyyli muistuttaa vahvasti Should.js:n syntaksia ja expect-tyyli vastaavasti Expect.js:n syntaksia. Koodikatkelmassa 18 on esimerkit kaikista kolmesta Chain assertiotyylistä.

```
var chai = require('chai');

var user = {
```

```
  name: 'John',
  pets: ['Poppy', 'Bella', 'Charlie', 'Daisy', 'Alfie']
};

// Chai - Assert syntax
var assert = chai.assert;
assert.property(user, 'name');
assert.typeOf(user.name, 'string');
assert.lengthOf(user.name, 4)
assert.equal(user.name, 'John');
assert.property(user, 'pets');
assert.lengthOf(user.pets, 5);

// Chai - Should syntax
var should = chai.should();
user.should.have.property('name');
user.name.should.be.a('string').and.have.lengthOf(4);
user.name.should.equal('John');
user.should.have.property('pets').with.lengthOf(5);

// Chai - Expect syntax
var expect = chai.expect;
expect(user).to.have.property('name');
expect(user.name).to.be.a('string').and.have.lengthOf(4);
expect(user.name).to.equal('John');
expect(user).to.have.property('pets').with.lengthOf(5);
```

Koodikatkelma 18. Esimerkki Chai-assertiokirjaston eri syntaksien käytöstä.

Huomattavaa Chain eri assertiotyyleissä on, että assert- ja expect-tyylien require-lauseissa vain viitataan Chain assert- ja expect-funktioihin, kun taas should-tyylin require-lause suorittaa should-funktion. Should-funktio luo jatkeen Object.prototype-oliolle, jotta JavaScript-olioista voidaan kutsua should-funktiota assertioketjun aloittamiseksi.

4.3. Angular-yksikkötestiajurit

Angular-sovellusten testien ajamiseen voidaan käyttää monia eri yksikkötestiajureita. Nämä testiajurit automatisoivat yksikkötestien ajamisen, mahdollisesti myös monissa eri verkkoselaimissa.

Ajaakseen testejä ilman testiajuria, testaajan tulisi itse:

1. Avata verkkoselain.
2. Navigoida testattavalle verkkosivulle.
3. Ajaa testimetodit.
4. Itse varmistaa tulosten oikeellisuus, eli testien läpäisy.
5. Tarvittaessa tehdä muutoksia koodiin.
6. Päivittää verkkoselaimen sivu ja ajaa testit uudestaan.

Testiajurin avulla voidaan suorittaa kaikki testit yhdellä komennolla ja nähdä niiden tulokset kerralla, ilman yllä listattuja vaiheita. Testiajuri voidaan myös integroida osaksi jatkuvan integraation palvelinta, jolloin testit voidaan ajaa automaattisesti lähdekoodimuutosten jälkeen. [Haq 2017]

Karma on Googlen kehittämä yksikkötestiajuri JavaScript-testeille. Karma on Angularin kanssa oletusarvoisesti käytettävä testiajuri, sillä se sisältyy Angular CLI:n asennukseen. Karma mahdollistaa testien automatisoidun ajamisen monissa eri verkkoselaimissa ja laitteissa.

”On the AngularJS team, we rely on testing and we always seek better tools to make our life easier. That's why we created Karma - a test runner that fits all our needs. [Karma a]”

Karma on pääasiallisesti suunniteltu yksikkötestaamiseen, mutta sitä voidaan käyttää myös end-to-end-testaamiseen. Karman kehittäjät kuitenkin suosittelevat Protractoria end-to-end-testien ajamiseen. [Karma b]

Riippuen käytettävästä testikehyksestä, Angularin testaamiseen voidaan käyttää muitakin testiajureita kuin Angular CLI:n sisältämää Karmaa. Jest ja Mocha sisältävät omat sisäiset yksikkötestiajurinsa, joten näidenkään testikehysten kanssa ei tarvitse itse asentaa erillistä yksikkötestiajuria. Karmaa ja Jestin yksikkötestiajuria käsitellään myöhemmin esimerkkien avulla kohdissa 5.1.1 ja 5.2.

4.4. Angular-end-to-end-testiajurit

Kuten yksikkötesteissä, myös end-to-end-testaamisessa testien automatisointi on tärkeää, jotta testejä ei tarvitse ajaa ja niiden tuloksia tarkistaa manuaalisesti. End-to-end-testejä voidaan automatisoida end-to-end-testiajureiden avulla. Toisin kuin yksikkötestiajurit, end-to-end-testiajurit on tarkoitettu verkkoselaimen käyttöliittymän manipuloinnin automatisointiin, jotta selaimen toiminnot suoritettaisiin testeissä juuri kuten oikea käyttäjäkin oikeassa käyttötilanteessa. Tässä työssä käsitellään Puppeteeria [Puppeteer 2020], Cypressiä [Cypress c] ja Seleniumiin [Selenium] perustuvaa Protractoria [Protractor].

4.4.1. Selenium

Selenium on työkalu verkkosovellusten end-to-end-testaamisen automatisointiin. Selenium avaa testattavan verkkosivun selaimessa ja on vuorovaikutuksessa sen kanssa, kuten oikeakin käyttäjä olisi.

”Selenium automates browsers. That's it! [Selenium]”

Seleniumilla voi automatisoida monia eri selaimia, kuten Google Chromea, Firefoxia, Edgeä ja Safaria. Seleniumia voi myös käyttää monilla eri ohjelmointikielillä, kuten Javalla, Pythonilla, C#:lla, Rubyllä ja JavaScript:llä. [Selenium 2020]

4.4.2. Protractor

Protractor on Googlen kehittämä ja Seleniumiin pohjautuva end-to-end-testiajuri juuri Angular-sovellusten testaamista varten. Protractor sisältyy Angular CLI:n asennukseen, joten sitä ei tarvitse asentaa erikseen. Protractor on rakennettu Seleniumin päälle lisäten Angular-kohtaisia elementtejä Angular-sovellusten testaamisen helpottamiseksi. Vaikka Protractor onkin luotu Angular-testaamiseen, voi sitä käyttää muidenkin JavaScript-sovellusten testaamiseen. Kuten Seleniumkin, myös Protractor avaa testattavan ohjelmiston verkkoselaimessa ja on vuorovaikutuksessa sivun kanssa oikean käyttäjän tavoin.

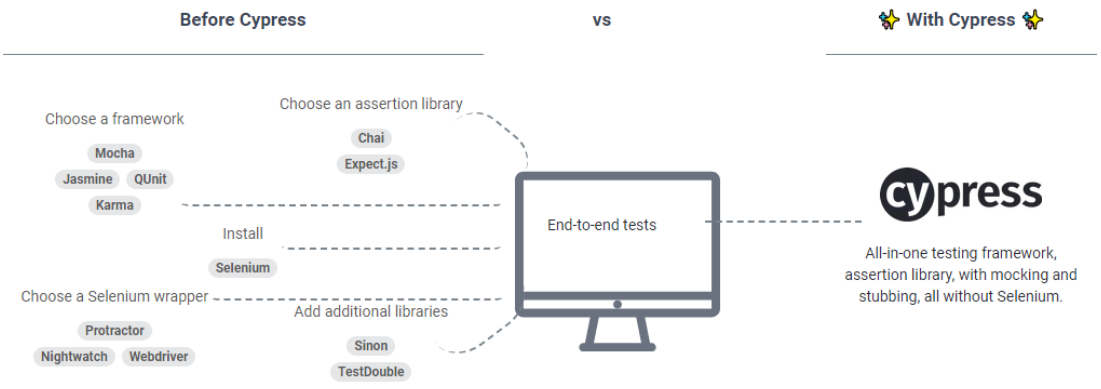
4.4.3. Puppeteer

Puppeteer on Googlen kehittämä rajapinta Google Chrome ja Chromium-selainten automatisointiin. Puppeteer ei sisällä testiajuria, joten Puppeteerin kanssa kannattaa käyttää testiajuria kuten Protractoria Angular-end-to-end-testien ajon automatisoimiseksi. Oletusarvoisesti Puppeteer toimii päättömänä verkkoselaimena, mutta sillä voidaan ajaa myös kokonaista Chrome-selainta.

Toisin kuin Protractor, Puppeteer ei käytä Seleniumia verkkosivujen manipulointiin. Puppeteer manipuloi Google Chrome -selainta suoraan ilman Seleniumia, jonka ansiosta Puppeteer on nopeampi kuin Selenium-pohjaiset ratkaisut, mutta se toimii vain Chrome-selaimessa testaamiseen. Puppeteer-testit toimivat vain JavaScriptillä kirjoitettuina Node.js-prosesseina, kun taas Selenium tukee useita eri ohjelmointikieliä.

4.4.4. Cypress

Cypress on ”kaiken sisältävä” end-to-end-testikehys. Yleensä kaiken kattavaan end-to-end-testaamiseen tarvittaisiin testikehys, end-to-end-testiajuri ja mahdollisesti erillinen assertiokirjasto sekä kirjastot tynkiä ja mock-olioita varten. Cypress sisältää kaikki nämä elementit yhden testikehyksen sisällä. [Cypress a]



Kuva 9. Cypress vastaan yksittäiset testityökalut. [Cypress a]

Kuvassa 9 havainnollistetaan, kuinka Cypress yksinkertaistaa JavaScriptin end-to-end-testaamista sisällyttämällä kaikki tarvittavat työkalut yhteen pakettiin. Cypressin ansiosta ei tarvitse asentaa monia eri työkaluja erikseen end-to-end-testaamista varten, eikä

kehittäjän siten tarvitse itse tutkia eri työkaluvaihtoehtoja ja niiden mahdollisia keskinäisiä yhteensopivuusongelmia.

Joitain Cypressin sisältämiä työkaluja ovat:

1. Mocha-testikehys: Cypress-testit siis käyttävät Mochan testisyntaksia.
2. Chai-assertiokirjasto: Chai mahdollistaa assertioiden kirjoittamisen monilla eri assertiotyyleillä.
3. Sinon.js-kirjasto: Sinon.js mahdollistaa testivakoojien, tynkien ja mock-olioiden käytön Cypress-yksikkötesteissä ja -integraatiotesteissä.
4. Sinon-Chai-kirjasto: Sinon-Chai mahdollistaa Chai-assertioiden kirjoittamisen Sinon-tynkiin ja -testivakoojiin liittyen. Sinon-Chai siis integroi Chai-assertiokirjaston ja Sinon.js:n yhtenäisemmäksi kokonaisuudeksi. [Cypress 2020]

Seleniumin sijasta Cypress toimii uudella varta vasten rakennetulla arkkitehtuurilla. Cypressin arkkitehtuuri toimii samassa ajosilmukassa kuin testattavan verkkosovelluksen koodi. Kuten Puppeteer-testejäkin, myös Cypress-testejä voi kirjoittaa vain JavaScript-kielellä. Cypress hallitsee koko automaatioprosessia sen alusta loppuun, jonka ansiosta se erottuu muiden testityökalujen joukosta. Koska Cypress toimii testattavan sovelluksen sisällä, sillä on suora pääsy testattavaan selainikkunaan, käyttöliittymäelementteihin, testattavan sovelluksen instanssiin ja kaikkiin muihin sovelluksen sisältämiin elementteihin. [Cypress a]

4.5. Päättömät verkkoselaimet

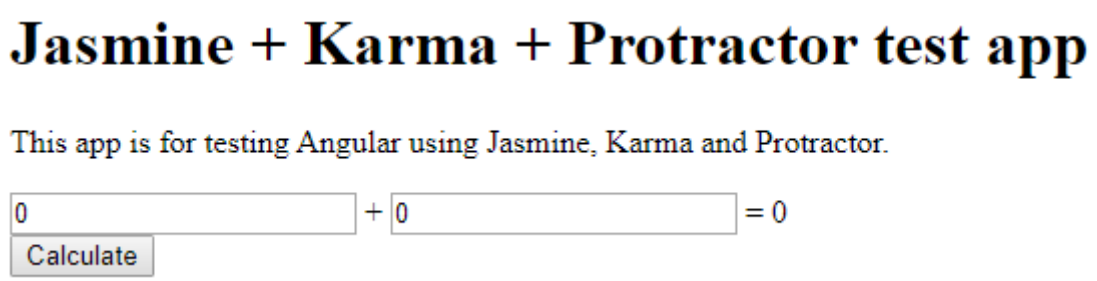
Päättömät verkkoselaimet ovat verkkoselaimia, joilla ei ole graafista käyttöliittymää. Päättömät verkkoselaimet mahdollistavat end-to-end-testien suorittamisen päättömässä tietokoneessa, kuten jatkuvan integraation palvelimella, jolla ei ole graafista käyttöliittymää.

PhantomJS on skriptattava päätön verkkoselain, eli selain ilman graafista käyttöliittymää. Alun perin PhantomJS:n julkaisi Ariya Hidayat vuonna 2011. PhantomJS käyttää QtWebKit-moottoria verkkosivujen ajamiseen [PhantomJS 2018]. Vuonna 2017 PhantomJS:n ylläpitäjä Vitaly Slobodin luopui asemastaan [Slobodin 2017] ja vuonna 2018 PhantomJS:n kehitys keskeytettiin toistaiseksi [GitHub 2018b]. PhantomJS:n kehitys on vieläkin keskeytetyssä tilassa, joten muitakin vaihtoehtoja päättömäksi verkkoselaimeksi verkkosovellusten testejä varten kannattaa siis miettiä.

Päättömien verkkoselainten kirjo onkin onneksi lisääntynyt merkittävästi PhantomJS:n julkaisun jälkeen. Jotkin tavalliset verkkoselaimet ovat saaneet päättömän tilan, joka mahdollistaa saman selaimen ajamisen, mutta ilman graafista käyttöliittymää. Esimerkiksi Mozilla Firefox sai päättömän tilan vuonna 2017 version 56 mukana [Mozilla 2019], samoin Google Chrome vuonna 2017 versiossa 59 [LePage 2019].

5. Angular-testityökalujen käyttöesimerkkejä

Tässä luvussa käydään läpi käyttöesimerkkejä monista Angular-ohjelmistojen testaamiseen käytetyistä testityökaluista. Testiesimerkit aloitetaan Jasminesta, Karmasta ja Protractorista. Nämä kolme työkalua ovat luonnollinen valinta testityökaluiksi, sillä ne sisältyvät Angular CLI:n asennukseen ja siten ovat käyttövalmiita heti Angularin asennuttua. Testattavana Angular-sovelluksena käytetään kuvassa 10 näkyvää yksinkertaista yhteenlaskusovellusta.



Kuva 10. Jasminella, Karmalla ja Protractorilla testattavan Angular-sovelluksen ulkoasu.

5.1. Jasmine-esimerkkejä

Tässä kohdassa käsitellään testiesimerkkejä, joissa käytetään Jasmine-testikehystä. Näissä esimerkeissä Angular-testaamiseen Jasminen kanssa on käytetty Karmaa, Protractoria ja Puppeteeria.

5.1.1. Jasmine + Karma -esimerkki

Tässä osassa käydään läpi Jasminen ja Karman yhteistoimintaa testaamalla Angular-komponenttia yksikkötesteillä. Koodikatkelmassa 19 on esimerkki Angular-komponentista. Komponentti sisältää yhden metodin Add, joka lisää kaksi parametreina saamaansa numeroa yhteen ja palauttaa niiden summan. Koodikatkelmasta voi myös huomata luvun 4 alussa mainitun TypeScriptin tarjoaman käännöksenaikaisen staattisen tyypityksen, sillä Add-metodin molemmat parametrit on vahvasti tyypitetty number-tyyppisiksi.

```
export class AppComponent {
  title = 'Jasmine + Karma + Protractor test app';

  Add(number1: number, number2: number) {
    if (number1 == null || number2 == null) {
      throw new Error("Can't add null!");
    }
    let result = number1 + number2;
    return result;
  }
}
```

Koodikatkelma 19. Angular-komponentti ja -metodi.

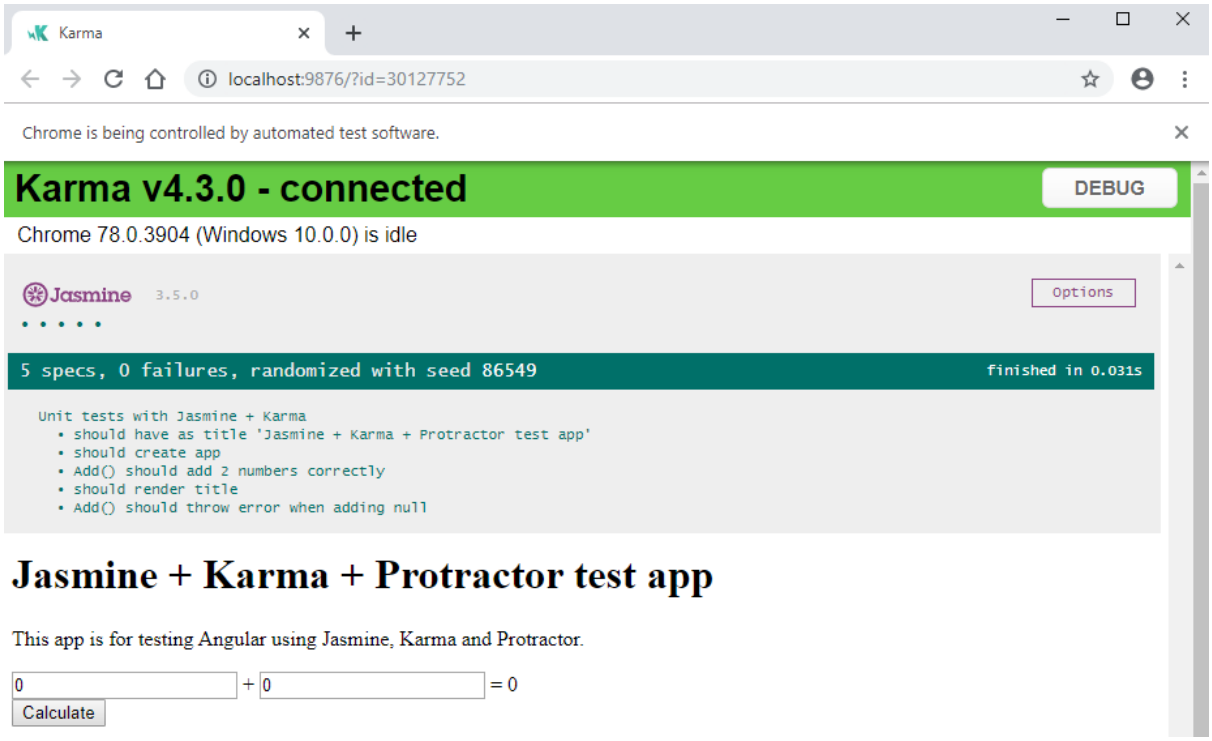
Koodikatkelmassa 20 on Jasmine-testisarja, joka sisältää viisi testiä. Describe-lohko kuvaa Jasminessa testisarjaa. Ensimmäinen parametri on testisarjan otsikko ja toinen on funktio, joka sisältää yksittäiset testit. It-lohko kuvaa yksittäistä testiä. Myös yksittäisellä testillä ensimmäinen parametri on testin otsikko ja toinen on funktio, joka sisältää testin vaiheet. Ensimmäinen testi nimeltään ”should create app” testaa sovelluskomponentin luomista. Seuraavat kaksi testiä testaavat sivun otsikon oikeellisuutta ja otsikon näytölle piirtymistä. Viimeiset kaksi testiä testaavat Add-metodin toimintaa. Kaksi numeroarvoa tulisi laskea yhteen, kun taas null-arvoa lisättäessä metodin tulisi heittää virhe.

```
describe('Unit tests with Jasmine + Karma', () => {
  it('should create app', () => {
    // Arrange + Act
    const app = new AppComponent();

    // Assert
    expect(app).toBeTruthy();
  });

  it(`should have as title 'Jasmine + Karma + Protractor test app'`, () => {
    // Arrange
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.componentInstance;

    // Act
    const title = app.title;

    // Assert
    expect(title).toEqual('Jasmine + Karma + Protractor test app');
  });

  it('should render title', () => {
    // Arrange
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.nativeElement;

    // Act
    const text = compiled.querySelector('h1').textContent;

    // Assert
    expect(text).toContain('Jasmine + Karma + Protractor test app');
  });

  it('Add() should add 2 numbers correctly', () => {
    // Arrange
    const calc = new AppComponent();
    const number1 = 1;
```

```
    const number2 = 2;
    const expected = 3;

    // Act
    const actual = calc.Add(number1, number2);

    // Assert
    expect(actual).toEqual(expected);
  });

  it('Add() should throw error when adding null', () => {
    // Arrange
    const calc = new AppComponent();
    const number1 = 1;
    const number2 = null;

    // Act + Assert
    expect( function(){ calc.Add(number1, number2); } )
      .toThrow(Error("Can't add null!"));
  });
});
```

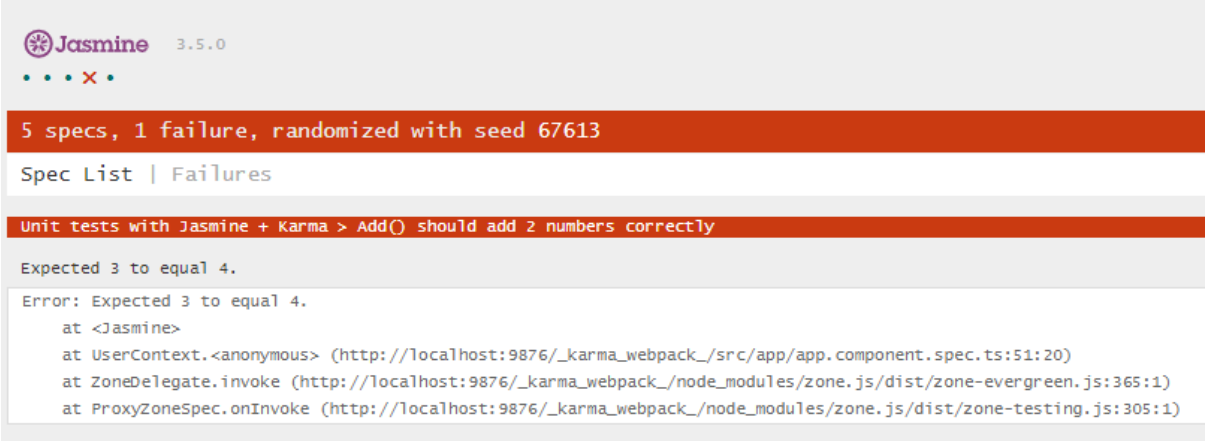
Koodikatkelma 20. Jasmine-testisarja, joka sisältää viisi testiä.

Testit voidaan ajaa konsolikomennolla ”ng test”, jolloin Karma ajaa kaikki löytämänsä testit. Tällöin uuteen selainikkunaan aukeaa kuvan 11 mukainen näkymä. Oletusarvoisesti Karma ajaa testit satunnaisessa järjestyksessä, joten testien tulisi olla eristettyjä toisistaan, jotta testien ajojärjestys ei vaikuttaisi testituloksiin.



Kuva 11. Karman antamat testitulokset.

Karma löysi koodikatkelmassa 20 annetun testisarjan ja ajoi sen sisältämät testit. Karma ilmoittaa nolla virhettä testeissä, joten kaikki testit menivät läpi. Jos lähdekoodi-tiedostoja käy muokkaamassa, Karma ajaa testit uudestaan automaattisesti niin kauan kuin ”ng test”-komennolla aloitettu komentoriviprosessi pidetään käynnissä. Oletusarvoisesti Karma ajaa testit Google Chrome -selaimessa, mutta selaimen pystyy myös vaihtamaan tai testit voi ajaa monessakin eri selaimessa. Tuettujen verkkoselainten lista ja ohjeet selaimen vaihtoon löytyvät Karman dokumentaatiosta [Karma c].



Kuva 12. Karman antama virheilmoitus testin epäonnistuessa.

Jos testi epäonnistuu, Karma näyttää kuvan 12 mukaisen virheilmoituksen. Virheilmoituksesta ilmenee virheen aiheuttanut testi ja virheen aiheuttaneen assertion koodirivi. Selainikkunan lisäksi Karma antaa vastaavan palautteen myös konsoliin, josta testi käynnistettiin.

5.1.2. Jasmine + Protractor -esimerkki

Tässä osassa käydään läpi Jasminen ja Protractorin yhteistoimintaa testaamalla Angular-komponenttia end-to-end-testeillä. Protractorilla ajettavat end-to-end-testit eroavat Karmalla ajettavista yksikkötesteistä siinä, että kun yksikkötesteissä testattavia metodeja kutsutaan suoraan, end-to-end-testeissä toimintoja testataan käyttöliittymän kautta. Testataan samaa kuvassa 10 esiteltyä Angular-sovellusta, mutta tällä kertaa end-to-end-testeillä käyttäen Protractoria.

```
import { AppPage } from './app.po';

describe('E2E tests with Jasmine + Protractor', () => {
  it('should display title \"Jasmine + Karma + Protractor test app\"', () => {
    let page = new AppPage();
    page.navigateTo();
    let title = page.getTitleText();
    expect(title).toEqual('Jasmine + Karma + Protractor test app');
  });
```

```
  it('should input and display 2 numbers', () => {
    let page = new AppPage();
    page.navigateTo();
    let number1 = 1;
    let number2 = 2;

    page.setFirstNumber(number1);
    page.setSecondNumber(number2);

    let firstNumber = page.getFirstNumber();
    let secondNumber = page.getSecondNumber();

    expect(firstNumber).toEqual(number1.toString());
    expect(secondNumber).toEqual(number2.toString());
  });

  it('should input 2 numbers, add them and display result', () => {
    let page = new AppPage();
    page.navigateTo();
    let number1 = 1;
    let number2 = 2;
    let expected = 3;

    page.setFirstNumber(number1);
    page.setSecondNumber(number2);

    page.clickCalculateButton();
    let result = page.getCalculationResult();

    expect(result).toEqual(expected.toString());
  });
});
```

Koodikatkelma 21. Jasmine + Protractor end-to-end-testisarja, joka sisältää kolme testiä.

Koodikatkelmassa 21 on Jasmine-testisarja, joka sisältää kolme end-to-end-testiä. Ensimmäinen testi testaa otsikon näyttämistä. Toinen testi testaa kahden luvun syöttämistä ja niiden näyttämistä. Kolmas testi testaa kahden luvun syöttämistä, niiden yhteenlaskemista ja tuloksen näyttämistä. Testeissä käytetty AppPage-luokka sisältää Protractor-toteutuksen, jonka kautta testeissä voidaan olla vuorovaikutuksessa sovelluksen käyttöliittymän kanssa.

```
import { browser, by, element } from 'protractor';

export class AppPage {
  navigateTo(): Promise<unknown> {
    return browser.get(browser.baseUrl) as Promise<unknown>;
  }
```

```
  getTitleText(): Promise<string> {
    return element(by.css('h1')).getText() as Promise<string>;
  }

  getFirstNumber(): Promise<string> {
    return element(by.id("firstNumber"))
      .getAttribute("value") as Promise<string>;
  }

  setFirstNumber(val: number) {
    element(by.id("firstNumber")).clear();
    element(by.id("firstNumber")).sendKeys(val);
  }

  getSecondNumber(): Promise<string> {
    return element(by.id("secondNumber"))
      .getAttribute("value") as Promise<string>;
  }

  setSecondNumber(val: number) {
    element(by.id("secondNumber")).clear();
    element(by.id("secondNumber")).sendKeys(val);
  }

  clickCalculateButton() {
    element(by.id("calculateButton")).click();
  }

  getCalculationResult(): Promise<string> {
    return element(by.id("resultField")).getText() as Promise<string>;
  }
}
```

Koodikatkelma 22. Protractor-metodeja sovelluksen käyttöliittymän käsittelyyn.

Koodikatkelman 22 Protractor-metodit ovat vuorovaikutuksessa sovelluksen HTML-elementtien kanssa. Metodit sisältävät getterit otsikkokentälle ja laskun tuloskentälle, setterit ja getterit lukujen syötekentille, sekä metodin yhteenlaskun suorittavan painikkeen klikkaamiselle. Protractor ajaa testit konsolikomennolla ”ng e2e”, tarkoittaen end-to-end-testejä. Onnistuneesta testien suorituksesta saadaan kuvan 13 kaltainen palaute.

```
DevTools listening on ws://127.0.0.1:58403/devtools/browser/6ef9da29-ed9f-493b-b8fd-9d5f11581272
Jasmine started

  E2E tests with Jasmine + Protractor
    √ should display title "Jasmine + Karma + Protractor test app"
    √ should input and display 2 numbers
    √ should input 2 numbers, add them and display result

Executed 3 of 3 specs SUCCESS in 2 secs.
[15:05:16] I/launcher - 0 instance(s) of WebDriver still running
[15:05:17] I/launcher - chrome #01 passed
```

Kuva 13. Protractorin antamat testitulokset.

Jos testeissä ilmenee virheitä, Protractor antaa virheilmoituksen kuvan 14 esimerkin mukaisesti. Tässä esimerkkivirheessä testi muokattiin odottamaan tulosta 4, vaikka oikea tulos on 3.

```
DevTools listening on ws://127.0.0.1:64556/devtools/browser/6871b123-1fc7-4d83-9ec1-c58622e6a575
Jasmine started

  E2E tests with Jasmine + Protractor
    √ should display title "Jasmine + Karma + Protractor test app"
    √ should input and display 2 numbers
    × should input 2 numbers, add them and display result
      - Expected '3' to equal '4'.
          at UserContext.<anonymous> (C:\RAD\Angular testit\angular-jasmine-protractor-test\e2e\src\app.e2e-spec.ts:44:20)

**************************************************
*                    Failures                    *
**************************************************

1) E2E tests with Jasmine + Protractor should input 2 numbers, add them and display result
  - Expected '3' to equal '4'.

Executed 3 of 3 specs (1 FAILED) in 3 secs.
[12:03:39] I/launcher - 0 instance(s) of WebDriver still running
[12:03:39] I/launcher - chrome #01 failed 1 test(s)
[12:03:39] I/launcher - overall: 1 failed spec(s)
[12:03:39] E/launcher - Process exited with error code 1
```

Kuva 14. Protractorin antamat testitulokset virheen ilmetessä.

5.1.3. Jasmine + Protractor + Puppeteer -esimerkki

Tässä esimerkissä end-to-end-testien selainrajapintana käytetään Puppeteeria. Testikehyksenä käytetään edelleen Jasminea ja Protractoria käytetään testien automatisoituun ajamiseen.

```
import * as puppeteer from 'puppeteer';

describe('Puppeteer e2e tests', () => {
  it('should display title \"Puppeteer test app\"', async () => {
    const browser = await puppeteer.launch();
    const page = await browser.newPage();
    await page.goto('http://localhost:4200');

    let title = await page.evaluate(() =>
      document.querySelector('h1').innerText);
    expect(title).toEqual('Puppeteer test app');
  });

  it('should input and display 2 numbers', async () => {
```

```
    const browser = await puppeteer.launch();
    const page = await browser.newPage();
    await page.goto('http://localhost:4200');

    let number1 = 1;
    let number2 = 2;

    await page.$eval('#firstNumber', (el, number) =>
      (el as HTMLInputElement).value = number, number1.toString());
    await page.$eval('#secondNumber', (el, number) =>
      (el as HTMLInputElement).value = number, number2.toString());

    let firstNumber = await page.$eval('#firstNumber', el =>
      (el as HTMLInputElement).value);
    let secondNumber = await page.$eval('#secondNumber', el =>
      (el as HTMLInputElement).value);

    expect(firstNumber).toEqual(number1.toString());
    expect(secondNumber).toEqual(number2.toString());
  });

  it('should input 2 numbers, add them and display result', async () => {
    const browser = await puppeteer.launch({
      headless: false
    });
    const page = await browser.newPage();
    await page.goto('http://localhost:4200');

    let number1 = 1;
    let number2 = 2;
    let expected = 3;

    await page.$eval('#firstNumber', (el, number) =>
      (el as HTMLInputElement).value = number, number1.toString());
    await page.$eval('#secondNumber', (el, number) =>
      (el as HTMLInputElement).value = number, number2.toString());

    await page.click('#calculateButton');
    let result = await page.$eval('#resultField', el =>
      (el as HTMLInputElement).value);

    expect(result).toEqual(expected.toString());
  });
});
```

Koodikatkelma 23. Jasmine + Puppeteer end-to-end-testisarja, joka sisältää kolme testiä.

Koodikatkelmassa 23 on Jasmine-testisarja, joka sisältää kolme end-to-end-testiä. Tämä testisarja suorittaa aiempaa Protractor-testiesimerkkiä vastaavat toimenpiteet,

mutta käyttäen Puppeteeria testien selainrajapintana Protractorin oman selainrajapinnan sijasta. Koska testit ajetaan käyttäen Protractoria, myös testipalautteet näytetään samalla tavalla kuin aiemmassa Jasmine + Protractor -esimerkissä.

5.2. Jest-esimerkki

Tässä kohdassa käydään läpi Jestin eroja Jasminen ja Karman yhdistelmään nähden esimerkin avulla. Koska Jest jakaa Jasminen syntaksin, koodikatkelmassa 20 esitetyn Jasmine-yksikkötestisarjan voi kopioida suoraan Jestiin, testejä ei tarvitse muuttaa.

```
PASS src/app/app.component.spec.ts
  Unit tests with Jest
    √ should create app (53ms)
    √ should have as title 'Jest test app' (46ms)
    √ should render title (27ms)
    √ Add() should add 2 numbers correctly (15ms)
    √ Add() should throw error when adding null (17ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        2.581s, estimated 4s
Ran all test suites.
```

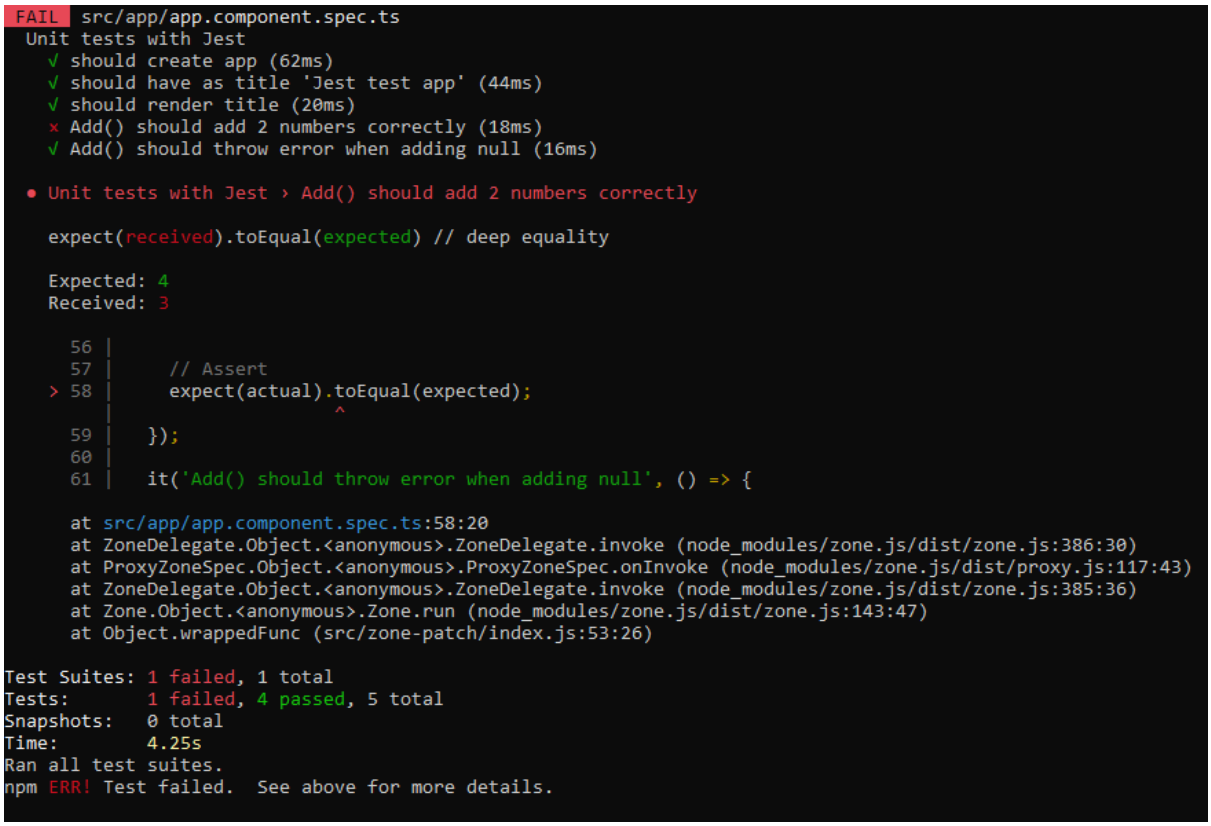
Kuva 15. Jestin testitulokset.

Kuvassa 15 on esimerkki Jestin antamista testituloksista. Karma ei omissa tuloksissaan ilmoita koko testiprosessiin kulunutta aikaa, mutta ainakin tässä työssä esitetyn esimerkin tapauksessa on silminnähden selvää, että Jest suorittaa testit nopeammin laskien testikomennon syöttämishetkestä testitulosten ilmestymiseen. Karman suhteelliselle hitaudelle voidaan löytää monia mahdollisia syitä. Karman täytyy joka testiajon yhteydessä kääntää ja rakentaa ohjelma lähdekoodista, joka hidastaa testien suorittamista. Jestin ei tarvitse tehdä tätä. Karman täytyy myös avata selainikkuna ja ajaa testit selaimessa, joka luultavasti hidastaa testien suoritusta.

Oikean selaimen sijasta Jest käyttää JSDOM:ia testien ajamiseen. JSDOM pyrkii emuloimaan verkkoselaimia juuri tarpeeksi, että Node.js-sovelluksia voidaan testata JSDOM:lla oikean verkkoselaimen sijasta [Jsdom 2020]. JSDOM on toteutettu JavaScript-kielellä ja se mahdollistaa web-standardien kuten DOM ja HTML käytön Node.js:ssä ilman oikeaa selainta. JSDOM on nopeampi kuin oikea selain, mutta JSDOM:lla ei voi testata selainkohtaisia eroja, joita voidaan mahdollisesti kohdata eri verkkoselaimia käytettäessä.

Jest myös tarjoaa yksityiskohtaisemman palautteen, kun testi epäonnistuu. Kuvassa 16 on esimerkki Jestin antamasta virheilmoituksesta testin epäonnistuessa. Jest tulostaa rivin, jolla virhe ilmeni, assertoitavat muuttujat ja niiden arvot, sekä virheen aiheuttaman

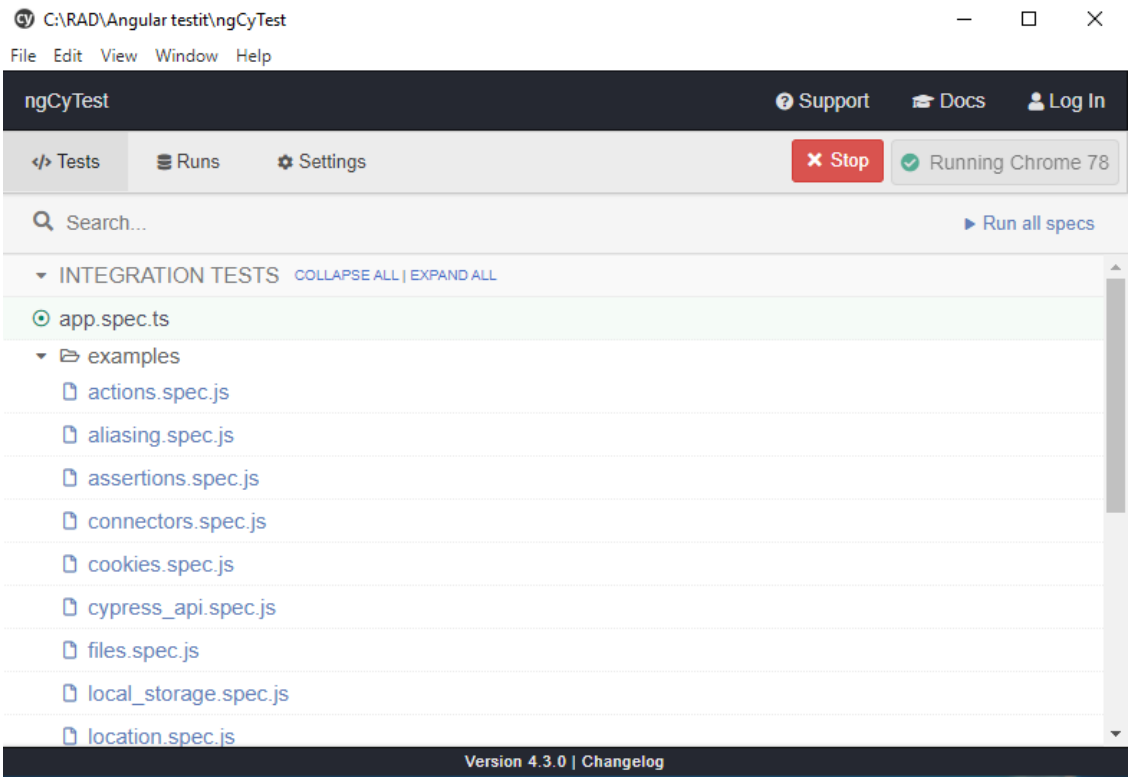
rivin edeltäviä ja seuraavia rivejä, joka helpottaa virheen kontekstin hahmottamista jo testin palautetta lukiessa ennen lähdekooditiedoston avaamista.



Kuva 16. Jestin antama virheilmoitus testin epäonnistuessa.

5.3. Cypress-esimerkki

End-to-end-testaaminen Cypressillä toimii hieman eri tavalla kuin Protractor ja Puppeteer. Cypress käynnistetään komentoriviltä, jolloin se aukeaa kuvan 17 näköiseen omaan sovellusikkunaansa. Cypressin käyttöliittymästä voi esimerkiksi käynnistää haluamansa testit, tarkastella aiemmin nauhoitettuja testiajoja ja muuttaa asetuksia. Cypress-testit voidaan myös ajaa suoraan komentoriviltä ilman käyttöliittymää.



Kuva 17. Cypress-sovellusnäkymä.

Jotkin Cypressin käyttöliittymän tarjoamista ominaisuuksista tosin vaativat Cypressin maksullisen Cypress Dashboard -palvelun, jonka avulla voi esimerkiksi ajaa useita Cypress-testejä samanaikaisesti, nähdä testiajoihin liittyvää analytiikkaa tai nähdä videoita epäonnistuneista testiajoista. [Cypress b]

Koodikatkelmassa 24 on Cypress-testisarja, joka sisältää kolme Cypress-testiä. Nämä testit suorittavat samat toimenpiteet kuin aiemmat Protractor- ja Puppeteer-esimerkkitestit, mutta käyttäen Cypress-selainrajapintaa selaimen manipulointiin.

```
describe('Cypress Tests', function() {
  it('should display title \"Cypress test app\"', () => {
    cy.visit('http://localhost:4200/');
    cy.get('h1').invoke('text').should('equal', 'Cypress test app');
  });

  it('should input and display 2 numbers', () => {
    cy.visit('http://localhost:4200/');
    let number1 = 1;
    let number2 = 2;

    cy.get('#firstNumber').clear();
    cy.get('#firstNumber').type(number1.toString());
    cy.get('#secondNumber').clear();
    cy.get('#secondNumber').type(number2.toString());

    cy.get('#firstNumber').invoke('val').should('equal', number1.toString());
```

```
    cy.get('#secondNumber').invoke('val').should('equal', number2.toString());
  });

  it('should input 2 numbers, add them and display result', () => {
    cy.visit('http://localhost:4200');
    let number1 = 1;
    let number2 = 2;
    let expected = 3;

    cy.get('#firstNumber').clear();
    cy.get('#firstNumber').type(number1.toString());
    cy.get('#secondNumber').clear();
    cy.get('#secondNumber').type(number2.toString());

    cy.get('#calculateButton').click();
    cy.get('#resultField').invoke('text')
      .should('equal', expected.toString());
  });
});
```

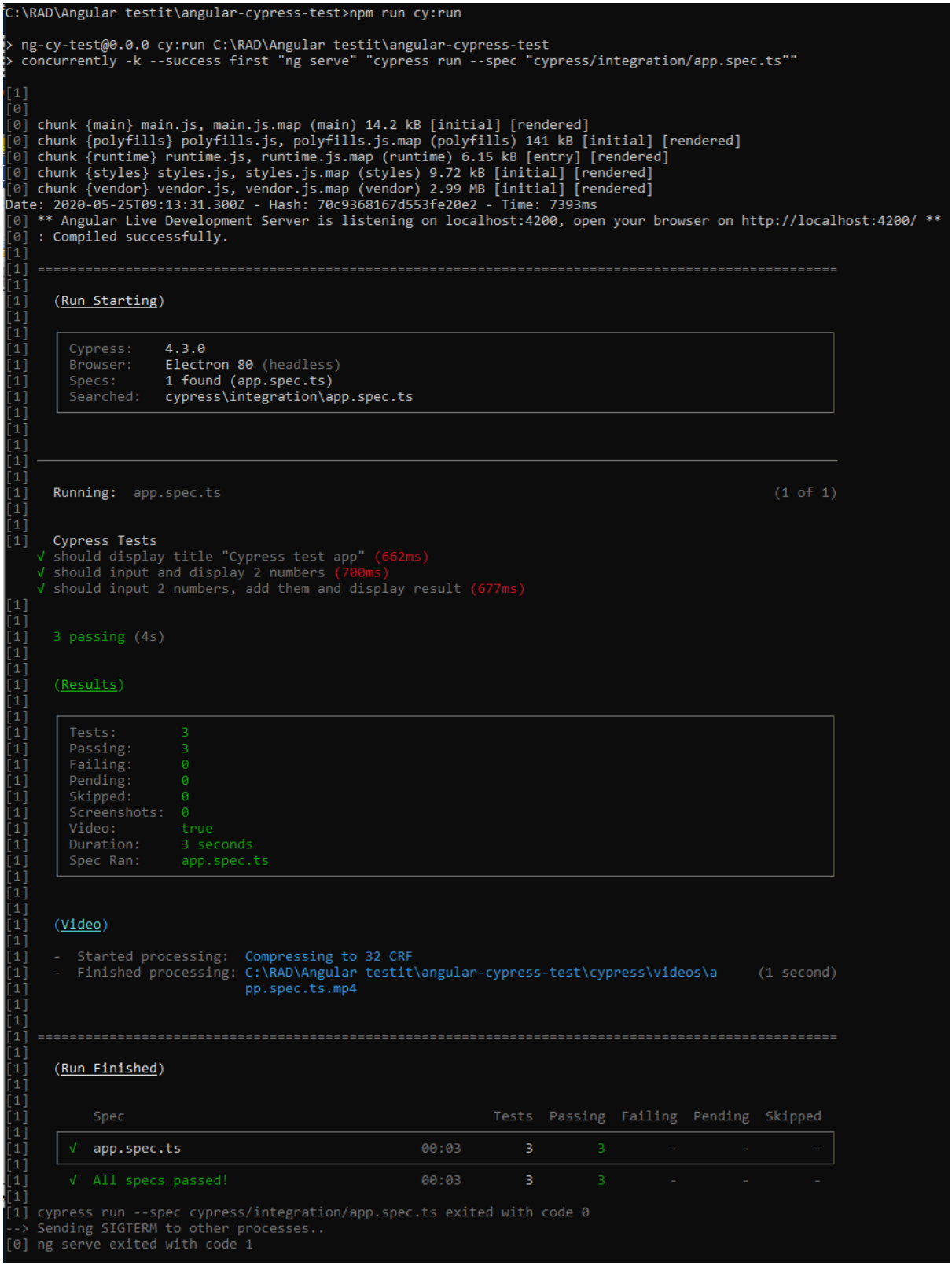
Koodikatkelma 24. Cypress end-to-end-testisarja, joka sisältää kolme testiä.

Toisin kuin Protractor, Cypress ei avaa Angular-palvelinta automaattisesti testien käynnistyessä. Angular-palvelin tulee manuaalisesti avata ennen Cypress-testien ajamista, jotta Cypress pääsee navigoimaan testattavan Angular-sovelluksen osoitteeseen. Cypress ei myöskään sulje Angular-palvelinta automaattisesti testien jälkeen, joten palvelin tulee myös sulkea manuaalisesti. Koodikatkelmassa 25 näkyvä node-komento ”cy:run” käynnistää Angular-palvelimen, ajaa Cypress-testit ja sulkee Angular-palvelimen Cypress-testien valmistuttua. Komennossa käytetään ”concurrently”-nimistä node-pakettia, joka mahdollistaa usean komennon ajamisen samanaikaisesti [Concurrently 2020]. Concurrently helpottaa Cypress-testien ajoa, sillä sen avulla Angular-palvelinta ei tarvitse manuaalisesti avata ja sulkea Cypress-testejä ajaessa. Jos end-to-end-testikehyksenä käytettäisiin Cypressin sijasta Protractoria, samoja ongelmia ei olisi Protractorin paremman Angular-integraation ansiosta. Protractor-testejä ajaessa Protractor käynnistää Angular-palvelimen automaattisesti ja myös sulkee sen testien loputtua.

```
{
  ...
  "scripts": {
    ...
    "cy:run": "concurrently -k --success first \"ng serve\" \"cypress run spec
              \"cypress/integration/app.spec.ts\"\""
  },
  ...
}
```

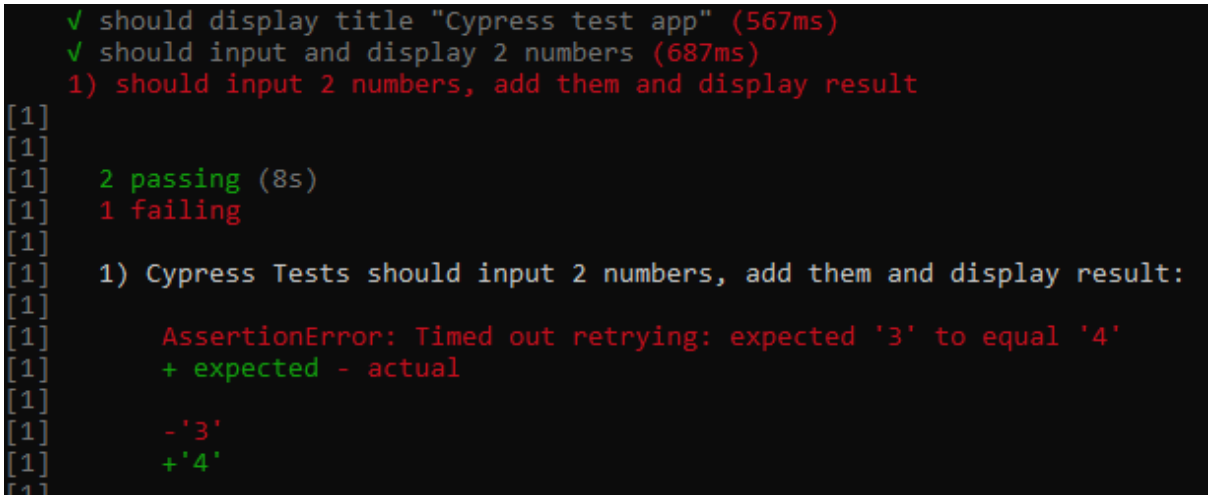
Koodikatkelma 25. Package.json-tiedostossa oleva komento Angular-palvelimen ja Cypress-testien ajamiselle samanaikaisesti.

Kuvassa 18 on esimerkki Cypress-testituloksista ja myös Angular-palvelimen sekä Cypressin ajosta yhdellä concurrently-komennolla. Testituloksista voi huomata, että Cypress automaattisesti tallentaa videon testisarjan uusimmasta ajosta.



Kuva 18. Concurrently-komennon avulla ajetun Cypress-testiajon tulokset.

Testin epäonnistuessa Cypress antaa testistä kuvan 19 mukaista palautetta. Tässä esimerkkivirheessä testi muokattiin odottamaan tulosta 4, vaikka oikea tulos on 3. Palautetta annetaan myös testisarjakohtaisesti ja testipalautteen lopussa on yhteenveto kaikista ajetuista testisarjoista, kuten näkyy kuvassa 18.



Kuva 19. Epäonnistuneen testin palaute Cypress-kehyksessä.

5.4. Robot Framework -esimerkki

Robot Framework on avoimen lähdekoodin geneerinen automaatiokehys, joka on suunnattu hyväksyntätestaukseen ja hyväksymistestausvetoiseen ohjelmistokehitykseen. Robot Frameworkin testitapauksissa käytetään avainsanapohjaista testausmenetelmää ja testitapaukset kirjoitetaan taulukkomuodossa. Robot Frameworkissa testitapaukset voidaan kuvailla luonnollisella kielellä, mikä on yksi suurimmista Robot Frameworkin eduista.

```
*** Settings ***
Documentation     Simple example using SeleniumLibrary.
Library           SeleniumLibrary

*** Variables ***
${WIKIPEDIA URL}  https://www.wikipedia.org/
${BROWSER}        Chrome

*** Test Cases ***
Valid Search
    [Tags]  Chrome
    Open Browser To Wikipedia
    Input Searchterm    John Deere
    Do Search
    John Deere Page Should Be Open
    [Teardown]    Close Browser

*** Keywords ***
Open Browser To Wikipedia
    Open Browser    ${WIKIPEDIA URL}    ${BROWSER}
    Title Should Be    Wikipedia

Input Searchterm
    [Arguments]    ${searchterm}
    Input Text    searchInput    ${searchterm}

Do Search
    Click Button    xpath=//*[@id="search-form"]/fieldset/button

John Deere Page Should Be Open
    Title Should Be    John Deere - Wikipedia
```

Koodikatkelma 26. Robot Frameworkilla luotu testi.

Koodikatkelman 26 testissä testataan Wikipedian verkkosivujen toimintaa Google Chrome -selaimessa. Verkkosivujen testaamisen Robot Frameworkin avulla mahdollistaa esimerkiksi Selenium, jota käytettiin tässä testissä verkkosivun käyttöliittymän manipulointiin. Testi avaa uuden Google Chrome -selainikkunan Wikipedian verkkosivuille, syöttää hakukenttään hakutermin ”John Deere”, tekee haun ja siten siirtyy John Deeren Wikipedia-sivulle, ja lopuksi sulkee selainikkunan. Tämän testin onnistuminen päätellään selainikkunan otsikon muutoksista, sillä Wikipedian etusivulla otsikko on ”Wikipedia” ja John Deeren Wikipedia-sivulla otsikko on ”John Deere - Wikipedia”.

6. Testauksen kehitys John Deere Forestry Oy:llä

Tässä luvussa käydään läpi John Deere Forestry Oy:n erään C#-ohjelmistoprojektin testaamiseen käytettävän testikehyksen sekä erään Angular-ohjelmistoprojektin testaamiseen käytettävien yksikkötestikehyksen, end-to-end-testikehyksen ja assertiokirjaston valintaprosessia. Toimin itse ohjelmistokehittäjänä molemmissa edellä mainituissa ohjelmistoprojekteissa.

6.1. C#-yksikkötestikehyksen valinta

Vertaillut vaihtoehdot C#-yksikkötestikehykseksi olivat MSTest, NUnit ja xUnit.net.

MSTestin etuna on sen sisältyminen Visual Studioon, joten se ei vaadi erillisiä asennustoimenpiteitä. Vertailun muut testikehykset tulee asentaa erikseen Visual Studion laajennoksina. Huonona puolena on MSTestin testiluokka-attribuutin vaadittavuus, joka tarkoittaa, että testejä sisältävät luokat täytyy merkitä erikseen testiluokiksi TestClass-attribuutilla. NUnitissa testiluokka-attribuutin käyttö on vapaaehtoista ja xUnit.netissä vastaavaa attribuuttia ei ole.

NUnitin etuina ovat sen sisältämät attribuutit Combinatorial, Sequential ja Pairwise, jotka helpottavat parametrisoitujen testiajojen generoinnissa. NUnitilla on myös vertailluista testikehyksistä parhaat viralliset dokumentaatiot, josta löytyy myös käyttöesimerkkejä NUnitin attribuuteille [NUnit 2020b]. Hyvän dokumentaation tärkeys korostuu varsinkin kehitystiimin kasvaessa, kun useamman kehittäjän pitää pystyä ymmärtämään ja kirjoittamaan testejä. MSTestille löytyy myös testirajapinnan dokumentaatiota [Microsoft 2018b; Microsoft 2020b; Microsoft b] ja käyttöesimerkkejä [Microsoft 2019b; Microsoft 2019c], mutta MSTestin dokumentaatio on melko hajautettua verrattuna NUnitin dokumentaatioon. xUnit.netin dokumentaatio on suppeaa muihin testikehyksiin verrattuna [xUnit.net b]. Käyttäjät ovat pyytäneet tarkempaa dokumentaatiota xUnit.netille, mutta kirjoitushetkellä xUnit.netin dokumentaatio on vertailun suppein [GitHub 2018d; GitHub 2019a; GitHub 2019b; GitHub 2019c; xUnit.net 2020].

xUnit.net pakottaa tiettyjä testauskäytäntöjä enemmän kuin muut vertailun testikehykset. xUnit.netissä ei ole testimetodien yhteisiä alustusmetodeja ollenkaan. Yhteiset alustusmetodit ovat yleensä huono idea, sillä niiden kautta testit saattavat tehdä tarpeettomia alustuksia. Toisaalta jos kaikissa luokan testeissä vaaditaan samat alustukset, yhteisillä alustusmetodeilla voidaan vähentää toistoa. Monet xUnit.netin alun perin uusista ominaisuuksista on myöhemmin lisätty myös MSTestiin ja/tai NUnitiin, kuten xUnit.netin Assert.Throws-metodi ja testiluokka-attribuutin poistaminen.

John Deere Forestry Oy:n C#-projektissa käytettäväksi C#-yksikkötestikehykseksi valittiin NUnit sen dokumentaation määrän ja laadun sekä ylimääräisten testiparametriattribuuttien takia. NUnitin syntaksi koettiin myös yleisesti selkeimmäksi.

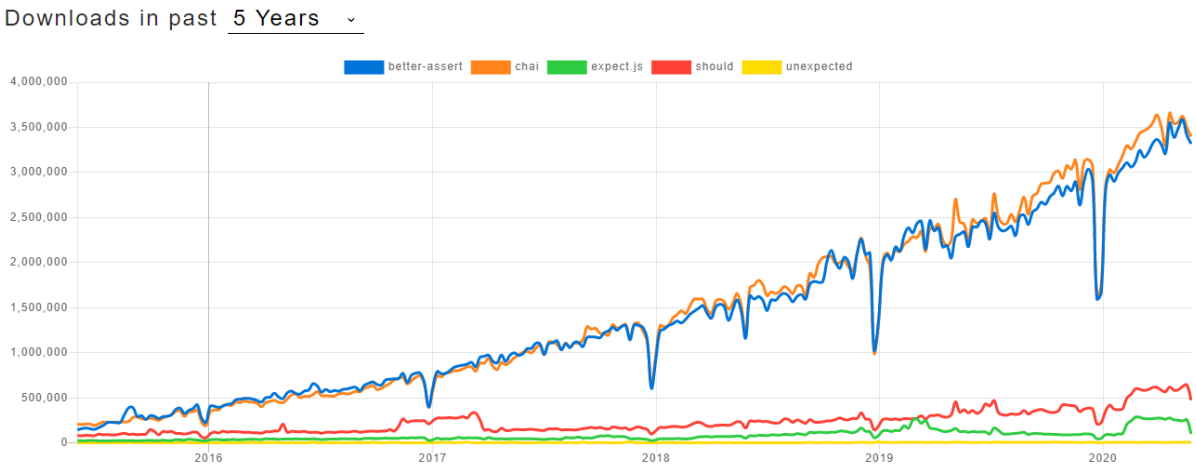
6.2. Angular-testityökalujen valinnat

Vertaillut vaihtoehdot Angular-testikehykseksi ja -yksikkötestiajuriksi olivat Jasmine Karman kanssa, Jest ja Mocha. Jasminen ja Karman etuna on niiden sisältyminen Angularin asennukseen, jonka vuoksi ne ovat oletusarvoiset yksikkötestityökalut Angularissa. Jestin etuina ovat sen nopeammat testiajot ja paremmaksi koetut testipalautteet. Mocha oli myös vaihtoehto testikehykseksi, mutta siitä kuitenkin luovuttiin koska sen käyttöönotossa Angular-projektin kanssa kohdattiin liikaa ongelmia.

Vertaillut vaihtoehdot Angular-end-to-end-testityökaluiksi olivat Protractor, Puppeteer ja Cypress. Protractorin etuna on sen sisältyminen Angularin asennukseen Jasminen ja Karman tavoin. Puppeteerin etuna Protractoriin verrattuna on jonkin verran nopeammat testiajot, mutta sillä voi testata vain Chrome-selaimessa. Cypressin etuina ovat sen sisältämät monet testityökalut ja testiajojen automaattinen videointi, jota voidaan hyödyntää epäonnistuneiden testien debuggauksessa. Cypressin testisyntaksi myös koettiin parhaaksi.

Vertaillut vaihtoehdot Angular-testaamisessa käytettäväksi assertiokirjastoksi olivat Assert, Better-assert, Should.js, Expect.js, Unexpected ja Chai. Assertiokirjastojen välillä on monia eroja, mutta Chai koettiin parhaaksi vaihtoehdoksi, sillä se tukee kaikkia kolmea assertiotyyliä: assert, should ja expect. Eri tyylit suorittavat lopulta samat toiminnot, mutta Chai antaa testien kehittäjille vaihtoehtoja käytettävän assertiotyylin suhteen.

NPM Trends -sivusto näyttää node-pakettien latausmäärät tietyn ajanjakson aikana. Kuvassa 20 näkyvien NPM Trends -tilastojen mukaan Chai ja Better-assert ovat selvästi ladatumpia assertiokirjastoja kuin Should.js, Expect.js ja Unexpected [NPM Trends 2020]. Latausmäärien ei tulisi olla ainoa syy jonkin assertiokirjaston valintaan, mutta se kannattaa myös ottaa huomioon. Jos kirjastolla on paljon käyttäjiä, sen kehitystä luultavasti tuetaan pidempään ja käytön avuksi voi löytää enemmän käyttäjien tuottamaa tukimateriaalia. Tilastojen mukaan Chain latausmäärät ovat selvästi olleet kasvussa viimeisten viiden vuoden aikana, joka viestii käyttäjien pitkäaikaisesta tyytyväisyydestä.



Kuva 20. Assertiokirjastojen latausmäärät viimeisten viiden vuoden aikana. [NPM Trends 2020]

John Deere Forestry Oy:n Angular-projektissa päädyttiin lopulta käyttämään Nrwl:n kehittämää Nx-työkalupakettia, joka sisältää valmiiksi Angularin, yksikkötestikehys Jestin sekä end-to-end-testikehys Cypressin. Nx suoraviivaistaa näiden testityökalujen asennusprosessia Angular-projektiin ja tekee niiden käyttämisestä helpompaa. Nx myös käynnistää Angular-palvelimen Cypress-end-to-end-testien ajamisen yhteydessä, siten poistaen tarpeen kohdassa 5.3 esitellyn concurrently-komennon käytölle.

6.3. Testauksen toteutus käyttäen jatkuvaa integraatiota

John Deere Forestry Oy:llä jatkuvan integraation palvelimen toteutukseen käytetään Jenkinsiä. Tässä kohdassa käydään läpi Jenkinsin toimintaa, käyttäen esimerkkinä Angular-sovelluksen Jest- ja Cypress-testien sekä C# .NET Core -sovelluksen NUnit-testien automaattista ajamista Jenkins-palvelimen avulla.

6.3.1. Jenkins Job DSL

Tässä esimerkissä käytetään Jenkinsin Job DSL -liitännäistä [Jenkins 2020a], jonka avulla voidaan yhden ”seed jobin” pohjalta generoida monia uusia jobeja. Seed job toimii käytännössä määrittelytiedostona uusille generoitaville jobeille, eikä se itsessään suorita muuta toiminnallisuutta. Tässä tapauksessa seed jobin avulla generoidaan yksi uusi job Angular-projektin testien ajamiselle ja toinen uusi job C#-projektin testien ajamiselle.

Aluksi Jenkinsiin luodaan uusi ”Freestyle project”-tyyppinen projekti, joka on nimeltään ”Seed Job”. Jobin nimen mukaisesti tämä tulee olemaan seed job. Jobiin lisätään uusi build-vaihe ”Process Job DSLs”, joka on saatu osana Job DSL -liitännäistä. Seed job suorittaa koodikatkelman 27 mukaisen DSL-skriptin. DSL-skriptin avulla seed job luo kaksi uutta jobia nimeltään ”Angular-project tests” ja ”CSharp-project tests”.

```
job('Angular-project tests') {
    scm {
        git {
            remote {
                url('Angular-project-SSH-key')
                credentials('GitHub-Angular-project-credentials')
            }
        }
    }
    triggers {
        githubPush()
    }
    steps {
        // Install node packages
        batchFile('call yarn install')

        // Run unit tests with Jest
        batchFile('call yarn test Angular-project')

        // Run end-to-end tests with Cypress
        batchFile('call yarn run e2e Angular-project-e2e –headless')
    }
    publishers {
        flowdock(FLOW_TOKEN_ANGULAR) {
```

```
            tags('jenkins', 'build')
        }
    }
}

job('CSharp-project tests') {
    scm {
        git {
            remote {
                url('CSharp-project-SSH-key')
                credentials('GitHub-CSharp-project-credentials')
            }
        }
    }
    triggers {
        githubPush()
    }
    steps {
        // Build C# application with MSBuild
        batchFile('dotnet build CSharp-project.sln')

        // Run unit tests with NUnit
        batchFile('dotnet test CSharp-project.sln')
    }
    publishers {
        flowdock(FLOW_TOKEN_CSHARP) {
            tags('jenkins', 'build')
        }
    }
}
```

Koodikatkelma 27. Jenkins seed jobin DSL-skripti.

6.3.2. Angular-testaus Jenkinsissä

Koodikatkelman 27 job nimeltään ”Angular-project tests” liitetään GitHub-varastoon, joka sisältää testattavan Angular-projektin. GitHub-varasto on määritetty lähettämään webhook-viestin Jenkins-palvelimelle aina kun GitHub-varastoon pusketaan muutoksia. DSL-skriptissä jobin rakentumisen laukaisimeksi määritetään GitHub-varastosta vastaanotetut webhook-viestit, joita varasto lähettää aina kun sinne pusketaan muutoksia. Kun Jenkins-palvelin vastaanottaa GitHub-varaston lähettämän webhook-viestin, varastoon liitetty Jenkins-job rakentuu automaattisesti. Jobin rakentumiselle saa asetettua muunkinlaisia laukaisimia, kuten ajastetusti tietyin väliajoin rakentamisen tai ajastetusti GitHub-varaston tarkistamisen muutosten varalta. Webhook kuitenkin todettiin tässä tapauksessa parhaaksi vaihtoehdoksi, sillä se takaa nopeimman palautteen aina koodimuutosten tapahtuessa, koska testiajot käynnistyvät Jenkinsissä välittömästi muutosten GitHub-varastoon puskemisen jälkeen. Job rakennetaan vain ja ainoastaan GitHubin webhook-viestin saapuessa, joten rakennus ei koskaan tapahdu ilman hyvää syytä, kuten voisi tapahtua, jos rakennus laukaistaisiin automaattisella ajastimella tietyin väliajoin tarkastamatta ensin GitHub-varaston tilaa.

Kun ”Angular-project tests”-job rakentuu, GitHub-varaston sisältö ladataan jobin työtilaan. Tämän jälkeen suoritetaan kolme eri Angular-projektin testaamiseen liittyvää

vaihetta. Aluksi asennetaan kaikki projektin tarvitsemat node-moduulit suorittamalla komento ”yarn install”. Sen jälkeen suoritetaan kaikki Angular-projektin sisältämät Jest-testit komennolla ”yarn test Angular-project”, joka tässä projektissa kääntyy komennoksi ”ng test Angular-project”, jolla ajetaan Angular-projektin yksikkötestit. Lopuksi projektin Cypressillä toteutetut end-to-end-testit ajetaan komennolla ”yarn run e2e Angular-project-e2e –headless”, joka kääntyy komennoksi ”ng e2e Angular-project -- --headless”. Headless-määrettä käyttämällä testit ajetaan Cypressin sisältämällä päättömällä Electron-selaimella.

6.3.3. C#-testaus Jenkinsissä

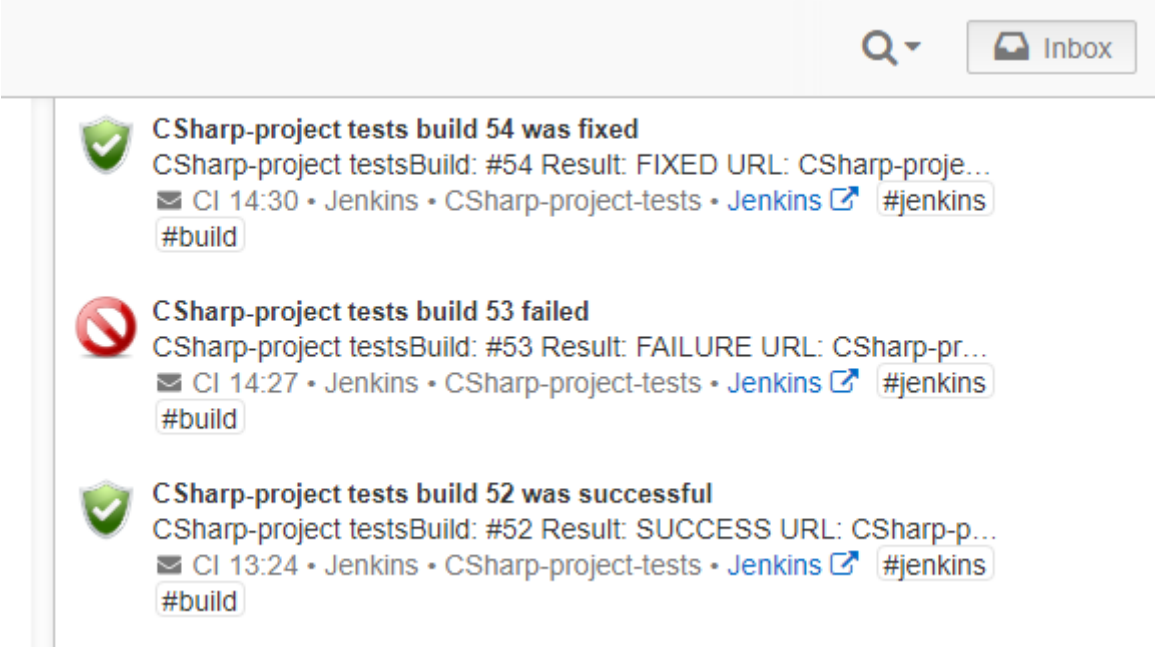
Koodikatkelman 27 job nimeltään ”CSharp-project tests” on Angular-jobin kaltaisesti liitetty GitHub-varastoon ja se on vastaavasti asetettu rakentumaan, kun se vastaanottaa webhook-viestin GitHub-varastosta. Testaus koostuu kahdesta osasta: ensin projekti rakennetaan ja sitten ajetaan yksikkötestit. Testattava C#-projekti on .NET Core -sovellus, joten sen rakentamiseen ja testien ajamiseen voidaan käyttää dotnet-komentoja. Komento ”dotnet build” rakentaa C#-projektin Jenkins-jobin työtilaan.

Testiajuriksi NUnit-testien ajamiseen jatkuvan integraation palvelimella oli kaksi vaihtoehtoa. Ensimmäinen vaihtoehto oli NUnit Console Runner: NUnitin kehittämä ohjelma NUnit-testien ajamiseen [NUnit 2017]. Ongelmaksi tämän testiajurin käytössä nousi .NET Core -tuen tarve, jota NUnit Console Runnerista ei vielä löydy. Tuki on kehitteillä, mutta ainakaan toistaiseksi sitä ei vielä löydy [NUnit 2020a; GitHub 2020; GitHub 2018c; Musser 2018]. Toinen vaihtoehto oli ”dotnet test”, joka on tarkoitettu juuri .NET Core -projektien testien ajamiseen [Microsoft 2020a]. Microsoft ja NUnit ehdottavat ”dotnet test”-komentoa yhtenä vaihtoehtona NUnit-testien ajamiseen .NET Core -projekteissa [Microsoft 2018a; NUnit 2018]. Dotnet test tukee myös monia muita C#-yksikkötestikehyksiä, joten käytettiinpä .NET Core -projektin testikehyksenä MSTestiä, NUnitia tai xUnit.netiä testit voidaan ajaa samalla komennolla.

6.3.4. Testipalaute kehittäjille

Testiajon valmistuttua Jenkinsissä testien tulokset voidaan lähettää suoraan kehittäjille. Nopean palautteen saaminen suoraan kehittäjien saataville on tärkeää, jotta mahdolliset virheet testiajoissa saadaan kehittäjien tietoon ja siten myös korjattua mahdollisimman nopeasti. Tulokset voidaan lähettää sähköpostilla esimerkiksi Jenkins Mailer-lisäosan [Jenkins 2020b] avulla tai ryhmäkeskustelupalvelu Flowdockiin [Flowdock a] Jenkinsin Flowdock-lisäosan [Jenkins 2015] avulla. Flowdockiin saa Jenkinsin testipalautteiden lisäksi integroitua monia muitakin ohjelmistokehityksessä käytettyjä työkaluja [Flowdock b], joka tekee Flowdockista hyvän tiedonjakokanavan laajoille ohjelmistoprojekteille, joissa on käytössä monia eri kehitystyökaluja. Molempien koodikatkelmassa 27 esiteltyjen jobien DSL-skriptien lopussa tehdään Flowdock-integraatio. Kun Jenkinsin Flowdock-lisäosa on liitetty Flowdockiin, se lähettää kuvan 21 mukaisia viestejä Flowdock-

keskusteluun, jossa kaikki keskustelun jäsenet voivat nähdä Jenkinsissä ajettujen testien tulokset yhdessä paikassa.



Kuva 21. Jenkinsissä ajettujen testien tuloksia Flowdock-keskustelussa.

7. Yhteenveto

Angular- ja C#-ohjelmistojen testaamiseen on kehitetty monia testityökaluja, jotka voivat suorittaa toisiinsa nähden täysin eri toimintoja tai joissain tapauksissa voivat erota toisistaan vain pienin syntaksieroin. Nämä kaikki työkalut kuitenkin pyrkivät osaltaan edistämään yhteistä lopullista tavoitetta: toimivaa ohjelmistoa.

Jokainen ohjelmistoprojekti on erilainen. Projektien vaatimukset ja tavoitteet ovat yksilöllisiä. Samoin projektien kehittäjät ja kehittäjien mieltymykset ovat yksilöllisiä. Vaihtoehtoiset testityökalut antavat kehittäjille mahdollisuuksia valita käytettävät työkalut projektien vaatimusten ja kehittäjien mieltymysten mukaisesti. Muiden kehittäjien antamat työkalusuositukset muuttuvat jatkuvasti työkalujen saadessa päivityksiä ja uusien työkalujen ilmestyessä, mutta se ei tarkoita, että yhdessä projektissa käytettäviä työkaluja tulisi vaihtaa toisiin aina uuden työkalun noustessa suosituimmaksi. Jos jokin testityökalu vaikuttaa sopivalta juuri sinun ohjelmistoprojektiisi ja projektin kehittäjät ovat tyytyväisiä siihen, ei välttämättä ole mitään syytä vaihtaa käytettäviä työkaluja toisiin.

”Use your brain to evaluate new things you bring into your stack, don't just follow the hype blindly. You don't have to switch your approach to how you handle CSS every week just because your Twitter stream indicates that is the way it should be done. [Simonsen 2016]”

Tässä työssä käsiteltiin ohjelmistotestaamisen automatisointia white box -tasolla. Aihetta käsiteltiin ensin yleisellä tasolla, jonka jälkeen esiteltiin joitain suosituimmista C#- ja Angular-testityökaluista. John Deere Forestry Oy:llä käytettäväksi C#-yksikkötestikehykseksi valittiin NUnit, Angular-yksikkötestikehykseksi valittiin Jest, Angular-end-to-end-testikehykseksi valittiin Cypress ja Angularin testaamisessa käytettäväksi assertiokirjastoksi valittiin Chai. Lisäksi tämän työn lopussa käsiteltiin C#- ja Angular-testien automaattista ajamista jatkuvan integraation palvelimella käyttäen Jenkinsiä.

Tässä työssä tehtyä ohjelmistotestauksen tutkimusta voisi vielä laajentaa muunkin kielisten ohjelmistojen testaamiseen, kuten esimerkiksi Python-, Java- tai PHP-ohjelmistojen testaamiseen. Samoin tutkittavia testaamisen muotoja voisi laajentaa sisältämään myös black box -tasolla testaamista, esimerkiksi käyttäjätestauksen muodossa.

8. Viiteluettelo

Angular. Testing. *https://angular.io/guide/testing* [Luettu 20.2.2020]

Beck, Kent. 1997. *Kent Beck's Guide to Better Smalltalk: A Sorted Collection*. Cambridge University Press.

Beck, Kent. 2002. *Test Driven Development: By Example*. Addison-Wesley Professional.

Black, Rex. 2011. *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. Wiley.

Concurrently. 2020. Concurrently. *https://www.npmjs.com/package/concurrently* [Luettu 29.4.2020]

Craig, Rick D., and Stefan P. Jaskiel. 2002. *Systematic Software Testing*. Artech House, Inc.

Cypress. 2020. Bundled Tools. *https://docs.cypress.io/guides/references/bundled-tools.html* [Luettu 6.4.2020]

Cypress. A. How it works. *https://www.cypress.io/how-it-works/* [Luettu 6.4.2020]

Cypress. B. Dashboard. *https://www.cypress.io/dashboard* [Luettu 27.4.2020]

Cypress. C. JavaScript End to End Testing Framework. *https://www.cypress.io/* [Luettu 11.10.2020]

Dahl, Ryan. 2009. Node version 0.1.22. *https://nodejs.org/docs/v0.1.22/api.html* [Luettu 11.3.2020]

Duvall, Paul M., Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional.

Flowdock. A. Features. *https://www.flowdock.com/features* [Luettu 19.5.2020]

Flowdock. B. Integrations. *https://www.flowdock.com/integrations* [Luettu 19.5.2020]

Fluin, Stephen. 2017. Branding Guidelines for Angular and AngularJS. *http://blog.angularjs.org/2017/01/branding-guidelines-for-angular-and.html* [Luettu 18.2.2020]

Garofalo, Raffaele. 2011. *Applied WPF 4 in Context*. Apress.

GitHub. 2012a. Add assert.defined, assert.undefined, assert.null, assert.notnull. *https://github.com/nodejs/node-v0.x-archive/issues/2879* [Luettu 11.3.2020]

GitHub. 2012b. C-style assert() messages. *https://github.com/nodejs/node-v0.x-archive/pull/2444* [Luettu 11.3.2020]

GitHub. 2013. Assert: display failing assert expression in assertion messages. *https://github.com/nodejs/node-v0.x-archive/pull/5761* [Luettu 11.3.2020]

GitHub. 2018a. Move the assert module to userland. *https://github.com/nodejs/node/issues/19652* [Luettu 11.3.2020]

GitHub. 2018b. Archiving the project: suspending the development. *https://github.com/ariya/phantomjs/issues/15344* [Luettu 24.2.2020]

GitHub. 2018c. Allow running of .NET Core assemblies via the .NET Framework 2.0 console/engine. *https://github.com/nunit/nunit-console/issues/478* [Luettu 13.5.2020]

GitHub. 2018d. Docs are horribly incomplete. *https://github.com/xunit/xunit/issues/1762* [Luettu 28.5.2020]

GitHub. 2019a. Documentation: List of available attributes and their uses. *https://github.com/xunit/xunit/issues/1900* [Luettu 28.5.2020]

GitHub. 2019b. Documentation: List of assertions and their meanings. *https://github.com/xunit/xunit/issues/1901* [Luettu 28.5.2020]

GitHub. 2019c. Documentation: Full API reference. *https://github.com/xunit/xunit/issues/1902* [Luettu 28.5.2020]

GitHub. 2020. Create .NET Core Console Runner. *https://github.com/nunit/nunit-console/issues/475* [Luettu 13.5.2020]

Greif, Sacha and Raphaël Benitte. 2019. The State of JavaScript 2019. *https://2019.stateofjs.com/* [Luettu 19.2.2020]

Haq, Md. Ziaul. 2017. *Angular Test-Driven Development - Second Edition*. Packt Publishing.

Hill, Mike and Joshua Kerievsky. Test-Driven Development: Evolving Simple Designs Guided By Tests — C# Edition. *https://elearning.industriallogic.com/gh/submit?Action=AlbumContentsAction&album=before&devLanguage=csharp* [Luettu 29.1.2020]

Jenkins. 2015. Flowdock plugin. *https://plugins.jenkins.io/jenkins-flowdock-plugin/* [Luettu 19.5.2020]

Jenkins. 2020a. Job DSL plugin. *https://plugins.jenkins.io/job-dsl/* [Luettu 28.4.2020]

Jenkins. 2020b. Mailer plugin. *https://plugins.jenkins.io/mailer/* [Luettu 19.5.2020]

Jenkins. Jenkins. *https://www.jenkins.io/* [Luettu 27.4.2020]

Jsdom. 2020. Jsdom. *https://www.npmjs.com/package/jsdom* [Luettu 30.3.2020]

Karma. A. Karma. *http://karma-runner.github.io/4.0/index.html* [Luettu 25.2.2020]

Karma. B. Frequently Asked Questions. *http://karma-runner.github.io/4.0/intro/faq.html* [Luettu 25.2.2020]

Karma. C. Browsers. *http://karma-runner.github.io/4.0/config/browsers.html* [Luettu 25.2.2020]

Khalid, Muhammad Adeel and Muhammad Ali Naeem. 2018. A surwey of automated testing tools. *VFAST Transactions on Software Engineering* 13, 1, 15-21.

LePage, Pete. 2019. New in Chrome 59. *https://developers.google.com/web/updates/2017/05/nic59* [Luettu 24.2.2020]

Microsoft. 2015. How to: Generate unit tests by using IntelliTest. *https://docs.microsoft.com/en-us/visualstudio/test/generate-unit-tests-for-your-code-with-intelli-test?view=vs-2019* [Luettu 13.1.2020]

Microsoft. 2018a. Unit testing C# with NUnit and .NET Core. *https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit* [Luettu 14.5.2020]

Microsoft. 2018b. Use the MSTest framework in unit tests. *https://docs.microsoft.com/en-us/visualstudio/test/using-microsoft-visualstudio-testtools-unittesting-members-in-unit-tests?view=vs-2019* [Luettu 28.5.2020]

Microsoft. 2019a. Run unit tests with Test Explorer. *https://docs.microsoft.com/en-us/visualstudio/test/run-unit-tests-with-test-explorer?view=vs-2019* [Luettu 29.1.2020]

Microsoft. 2019b. Unit test basics. *https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019* [Luettu 28.5.2020]

Microsoft. 2019c. Walkthrough: Create and run unit tests for managed code. *https://docs.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2019* [Luettu 28.5.2020]

Microsoft. 2020a. Dotnet test. *https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-test* [Luettu 13.5.2020]

Microsoft. 2020b. Docs for MSTest V2 test framework. *https://github.com/Microsoft/testfx-docs* [Luettu 28.5.2020]

Microsoft. A. Assert.ThrowsException Method. *https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert.throwsexception* [Luettu 28.1.2020]

Microsoft. B. Microsoft.VisualStudio.TestTools.UnitTesting Namespace. *https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting?view=mstest-net-1.3.2* [Luettu 28.5.2020]

Mocha. 2020. Assertions. *https://mochajs.org/#assertions* [Luettu 4.3.2020]

Mozilla. 2019. Firefox 56 for developers. *https://developer.mozilla.org/fi/docs/Mozilla/Firefox/Releases/56* [Luettu 24.2.2020]

Musser, Joseph. 2018. NUnit.ConsoleRunner does not run on .NET Core yet. *https://github.com/nunit/nunit/issues/2713#issuecomment-365943718* [Luettu 13.5.2020]

Newkirk, James. 2007a. Announcing xUnit.net. *https://jamesnewkirk.typepad.com/posts/2007/09/announcing-xuni.html* [Luettu 23.1.2020]

Newkirk, James. 2007b. Why you should not use SetUp and TearDown in NUnit. *https://jamesnewkirk.typepad.com/posts/2007/09/why-you-should-.html* [Luettu 27.1.2020]

Node.js. Assert. *https://nodejs.org/api/assert.html#assert_assert* [Luettu 3.3.2020]

NPM Trends. 2020. Better-assert vs Chai vs Expect.js vs Should.js vs Unexpected. *https://www.npmtrends.com/better-assert-vs-chai-vs-expect.js-vs-should-vs-unexpected* [Luettu 4.6.2020]

NUnit. 2009. NUnit 2.5 Final Release. *https://nunit.org/docs/2.5/releaseNotes.html* [Luettu 28.1.2020]

NUnit. 2017. Console Runner. *https://github.com/nunit/docs/wiki/Console-Runner* [Luettu 13.5.2020]

NUnit. 2018. .NET Core and .NET Standard. *https://github.com/nunit/docs/wiki/.NET-Core-and-.NET-Standard* [Luettu 14.5.2020]

NUnit. 2019. Assert.Throws. *https://github.com/nunit/docs/wiki/Assert.Throws* [Luettu 28.1.2020]

NUnit. 2020a. .NET Core support. *https://github.com/nunit/nunit-console/projects/1* [Luettu 13.5.2020]

NUnit. 2020b. NUnit Documentation. *https://github.com/nunit/docs/wiki/NUnit-Documentation* [Luettu 28.5.2020]

PhantomJS. 2018. PhantomJS - Scriptable Headless Browser. *https://phantomjs.org/* [Luettu 24.2.2020]

Protractor. Protractor - end-to-end testing for AngularJS. *https://www.protractortest.org/* [Luettu 11.10.2020]

Puppeteer. 2020. Puppeteer. *https://developers.google.com/web/tools/puppeteer* [Luettu 11.10.2020]

Rothermel, G., R. H. Untch, Chengyun Chu, and M. J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10, 929-948.

Sale, David. 2014. *Testing Python: Applying Unit Testing, TDD, BDD and Acceptance Testing*. John Wiley & Sons.

Selenium. 2020. Driver requirements. *https://www.selenium.dev/documentation/en/webdriver/driver_requirements/* [Luettu 31.3.2020]

Selenium. SeleniumHQ Browser Automation. *https://www.selenium.dev/* [Luettu 24.2.2020]

Simonsen, Sune. 2016. Unexpected - The Extensible BDD Assertion Toolkit - Interview with Sune Simonsen. *https://survivejs.com/blog/unexpected-interview/* [Luettu 12.3.2020]

Slobodin, Vitaly. 2017. [Announcement] Stepping down as maintainer. *https://groups.google.com/d/msg/phantomjs/9aI5d-LDuNE/5Z3SMZrqAQAJ* [Luettu 25.2.2020]

Stack Overflow. 2017. Node.js assert library vs. other assert libraries. *https://stackoverflow.com/a/39284536* [Luettu 3.3.2020]

Tsai, Yowko. 2019. xUnit.net.TestGenerator - Visual Studio Marketplace. *https://marketplace.visualstudio.com/items?itemName=YowkoTsai.xUnitnetTestGenerator* [Luettu 4.2.2020]

Wake, Bill. 2011. 3A – Arrange, Act, Assert. *https://xp123.com/articles/3a-arrange-act-assert/* [Luettu 29.1.2020]

Watkins, John and Simon Mills. 2011. *Testing IT: An Off-the-Shelf Software Testing Process*. Cambridge University Press.

xUnit.net. 2020. Where is xUnit API reference / in detail documentation? *https://github.com/xunit/xunit/issues/1018* [Luettu 28.5.2020]

xUnit.net. A. Home > xUnit.net. *https://xunit.net/* [Luettu 23.1.2020]

xUnit.net. B. Documentation. *https://xunit.net/#documentation* [Luettu 28.5.2020]